

# About this Reference

The following notations are used in this reference:

<b>KEYWORD</b>	Commands and language keywords.
<b><u>KEYWORD</u></b>	The default value for a command or language keyword when multiple values are possible but none are actually specified.
<a href="#">Phrase</a>	Typically indicates a hypertext link to a separate panel containing a description for that phrase.
<i>Parameter</i>	Parameters whose actual names or values are to be supplied by the programmer.
<b><i>Definition</i></b>	A term being defined for the first time, or special emphasis.
<b>Subscript</b>	Subscripted text.
<b>Superscript</b>	Superscripted text (other than <sup>2</sup> ).
<Name>	A text value represented by "Name" is to be substituted in place of <Name>, typically at assembler run-time.

-----

## Assembly Language Processor (ALP) Overview

The Assembly Language Processor (ALP) is an assembler that runs under OS/2 Warp. ALP is a functional replacement for the Microsoft Macro Assembler (MASM) and accepts:

- The full syntax of the Intel 80X86 architecture
- The full syntax of the MASM 5.10 high-level directive language
- A subset of the MASM 6.00 high-level directive language

ALP generates standard Object Module Format (OMF) files that can be linked to produce DOS or OS/2 executables. It can also generate symbolic debugging information compatible with the IBM family of source code debuggers. A MASM 5.10-compatible command line utility (MASM2ALP) is also provided to enable use of ALP with little or no change to existing build environments.

ALP also offers a rich set of command line options, as well as a comprehensive listing output capability that is highly configurable, allowing a visual perspective not possible with other assemblers.

-----

## Installation

The following components are part of the ALP package:

<b>ALP</b>	The assembler itself. These are the basic files that must be installed before the assembler can be used.
<b>MASM2ALP</b>	The MASM 5.10 Command Line Driver. The ALP component must be installed before the MASM2ALP utility can be utilized.

-----

## Installing ALP

ALP consists of two files:

alp.exe  
alp.msg

You can rename the root portions of the two file names if desired. In most cases, it does not matter whether the file names are in upper- or

lowercase because the default OS/2 file systems disregard case. It is possible, however, that the use of an OS/2 Installable File System (IFS) might require that file names be referenced exactly as they are named with respect to upper- and lowercase. If this is true, then the root portion of the **alp.exe** and **alp.msg** file names (see [BaseEXE](#)) must be spelled identically and the *.msg* extension on the messages file name must be specified in lowercase or the assembler will not be able to find the messages file at run time.

To install ALP on OS/2:

1. Copy **alp.exe** into a directory of your choice. If the assembler will be invoked from the command line (rather than by absolute reference from a makefile or command file), then the selected directory should be among those referenced by the [PATH](#) environment variable.
2. For best performance, the **alp.msg** file should be copied into the same directory used in step 1 or in a directory referenced by the [<BaseEXE>\\_PATH](#) environment variable. It is not necessary to set any additional environment variables if the first method is used.  
  
Alternatively, a directory referenced by the [DPATH](#) environment variable may also be used, but performance may be degraded during initialization, since the assembler must search all of the listed directories for the **alp.msg** file. See [The ALP Messages File](#) for more information.
3. Optionally, default values for command line options may be established. See [<BaseEXE>\\_OPTIONS](#) for more information.

---

## Installing MASM2ALP

The MASM2ALP utility consists of a single file:

`masm2alp.exe`

To install MASM2ALP on OS/2:

1. Copy **masm2alp.exe** into a directory of your choice. If the utility will be invoked from the command line (rather than by absolute reference from a makefile or command file), then the selected directory should be among those referenced by the [PATH](#) environment variable.  
  
It is recommended that both **masm2alp.exe** and **alp.exe** be installed in the same directory, especially in build environments where reliance on environment variables is discouraged. If **masm2alp.exe** is invoked by absolute path name (rather than via a [PATH](#) search), it will use that same path when it first attempts to execute **alp.exe**. This technique allows execution of **alp.exe** without requiring it to be referenced in the [PATH](#). If the search fails, the path name prefix will be removed and MASM2ALP will rely on the operating system to locate **alp.exe**.
2. If you also use the Microsoft Macro Assembler, you must decide if MASM2ALP will replace MASM or co-exist with it. This usually means renaming the MASM executable (**masm.exe**) to something else, then renaming **masm2alp.exe** to **masm.exe**. Alternatively, you may choose to leave the names unchanged, taking manual steps in your makefiles or build scripts to insure that the correct executables are referenced from your build environment.
3. Optionally, default values for command line options may be established by defining the **MASM** environment variable. MASM2ALP interprets the contents of this environment variable before processing the command line. Any values set in this manner are translated and passed to **alp.exe** via the command line. The [<BaseEXE>\\_OPTIONS](#) environment variable used by **alp.exe** is not queried or modified by MASM2ALP.
4. If it becomes necessary to analyze a problem with MASM2ALP (such as failure to invoke **alp.exe**, or other unexpected behavior), defining the **ALP\_ECHO** environment variable to any non-empty value will cause MASM2ALP to echo the generated command line to the standard output device.

---

## Understanding ALP

This chapter describes:

- The ALP message file
- ALP internal variables

Though you do not need to understand this information to be able to use ALP, you may need this information for troubleshooting purposes.

---

# The ALP Messages File

Nearly every message displayed by ALP at run time is stored in a separate message file. The exception to this rule are messages that are displayed if the message file cannot be opened: ALP ends if one of these messages is displayed.

When ALP starts, it determines the name of the message file by creating a name of the form **<BaseEXE>.msg** (see [BaseEXE](#)). Once ALP knows the name of the message file, ALP searches the following directories in the following order for the file:

1. Current directory
2. The directory contained in the [BasePATH](#) internal variable
3. Each of the directories specified in the [BaseEXE](#) internal variable
4. Each of the directories specified in the [PATH](#) environment variable
5. Each of the directories specified in the [DPATH](#) environment variable

---

## Internal Variables

ALP maintains a set of *internal variables* that it uses for various purposes. These variables reflect the ALP environment; programmers do not use these variables. Their values may be indirectly affected by the user of ALP, for instance, through the use of various command line options.

---

## BaseEXE

When ALP is invoked, if the full path name of the ALP executable was provided by the operating system, then the "base" portion is isolated and used to construct the value of the **BaseEXE** internal variable. For instance, if the user invoked ALP and the name of the executable was made available as "C:\TOOLS\ALP.EXE", then the **BaseEXE** internal variable would contain the value "ALP". The value of **BaseEXE** is used to differentiate ALP-specific components in the environment (such as data files or environment variables) from those that are globally accessible to all programs. Even multiple versions of ALP can coexist without environmental "collisions" simply by copying and renaming the ALP executable and its associated message file.

If the file name of the ALP executable is **not** available at run time, then the value of **BaseEXE** defaults to "ALP".

---

## BasePATH

When ALP is invoked, if the full path name of the ALP executable was provided by the operating system, then the "path" portion is isolated and used to construct the value of the **BasePATH** internal variable. For instance, if the user invoked ALP and the name of the executable was made available as "C:\TOOLS\ALP.EXE", then the **BasePATH** internal variable would contain the value "C:\TOOLS\". The value of **BasePATH** can be used to locate ALP-specific components in the environment (such as the ALP messages file) without the need to store this information in an alternate environment variable such as DPATH. Check your operating system documentation to see if it feasible to use the **BasePATH** method of locating ALP components.

If the file name of the ALP executable is **not** available at run time, then the value of **BasePATH** is NULL.

---

## DepDIR

This variable contains the empty string unless explicitly initialized with the **Fdd** parameterized command line option; it is only used if the value specified by the **Fd** command line option did not contain any path information.

Related Information:

- [Options](#)

- [File Names](#)
  - [Fd - Produce Make Dependency File](#)
  - [Fdd - Directory to Store Make Dependency File \(DepDIR\)](#)
- 

## DepEXT

This variable contains the default Make dependency file name extension that is conditionally appended to the concatenated values of [DepDIR](#) and [DepNAME](#); the assembler treats the resulting string as the fully qualified Make dependency file name. The default value for **DepEXT** is ".u" unless altered by use of the **Fed** parameterized command line option.

Related Information:

- [Options](#)
  - [File Names](#)
  - [Fed - Control Make Dependency File Extension \(DepEXT\)](#)
- 

## DepNAME

This variable contains the same value as the contents of [SrcNAME](#), unless initialized with the **Fd** parameterized command line option.

Related Information:

- [Options](#)
  - [File Names](#)
  - [Fd - Produce Make Dependency File](#)
- 

## IncDIR

This variable contains the empty string unless explicitly initialized with the **Fdi** parameterized command line option; it contains the cumulative value of all the specified include paths.

Related Information:

- [Options](#)
  - [File Names](#)
  - [Fdi - Specify Include File Search Path](#)
- 

## IncEXT

This variable contains the default include file name extension that is conditionally appended to unadorned file names generated by the **INCLUDE** preprocessor directive. The default value for **IncEXT** is ".inc" unless altered by use of the **Fei** parameterized command line option.

Related Information:

- [Options](#)
  - [File Names](#)
  - [Fei - Control Include File Extension \(IncEXT\)](#)
-

# LstDIR

This variable contains the empty string unless explicitly initialized with the **Fdl** parameterized command line option; it is only used if the value specified by the **FI** command line option did not contain any path information.

Related Information:

- [Options](#)
  - [File Names](#)
  - [FI - Produce Listing File](#)
  - [Fdl - Directory to Store Listing File \(LstDIR\)](#)
- 

# LstEXT

This variable contains the default listing file name extension that is conditionally appended to the concatenated values of [LstDIR](#) and [LstNAME](#); the assembler treats the resulting string as the fully qualified listing file name. The default value for **LstEXT** is ".lst" unless altered by use of the **Fel** parameterized command line option.

Related Information:

- [Options](#)
  - [File Names](#)
  - [Fel - Control Listing File Extension \(LstEXT\)](#)
- 

# LstNAME

This variable contains the same value as the contents of [SrcNAME](#), unless initialized with the **FI** parameterized command line option.

Related Information:

- [Options](#)
  - [File Names](#)
  - [FI - Produce Listing File](#)
- 

# MsgDIR

This variable contains the empty string unless explicitly initialized with the **Fdm** parameterized command line option; it is only used if the value specified by the **Fm** command line option did not contain any path information.

Related Information:

- [Options](#)
  - [File Names](#)
  - [Fm - Produce Messages File](#)
  - [Fdm - Directory to Store Messages File \(MsgDIR\)](#)
- 

# MsgEXT

This variable contains the default messages file name extension that is conditionally appended to the concatenated values of [MsgDIR](#) and

[MsgNAME](#); the assembler treats the resulting string as the fully qualified messages file name. The default value for **MsgEXT** is ".msg" unless altered by use of the **Fem** parameterized command line option.

Related Information:

- [Options](#)
- [File Names](#)
- [Fem - Control Messages File Extension \(MsgEXT\)](#)

-----

## MsgNAME

This variable contains the same value as the contents of [SrcNAME](#), unless initialized with the **Fm** parameterized command line option.

Related Information:

- [Options](#)
- [File Names](#)
- [Fm - Produce Messages File](#)

-----

## ObjDIR

This variable contains the empty string unless explicitly initialized with the **Fdo** parameterized command line option; it is only used if the value specified by the **Fo** command line option did not contain any path information.

Related Information:

- [Options](#)
- [File Names](#)
- [Fo - Produce Object File](#)
- [Fdo - Directory to Store Object File \(ObjDIR\)](#)

-----

## ObjEXT

This variable contains the default object file name extension that is conditionally appended to the concatenated values of [ObjDIR](#) and [ObjNAME](#); the assembler treats the resulting string as the fully qualified object file name. The default value for **ObjEXT** is ".obj" unless altered by use of the **Feo** parameterized command line option.

Related Information:

- [Options](#)
- [File Names](#)
- [Feo - Control Object File Extension \(ObjEXT\)](#)

-----

## ObjNAME

This variable contains the same value as the contents of [SrcNAME](#), unless initialized with the **Fo** parameterized command line option.

Related Information:

- [Options](#)
- [File Names](#)

- [Fo - Produce Object File](#)

-----

## SourceNAME

This variable contains the name of the top-level source file currently being processed by the assembler; its contents appear exactly as the user typed it on the command line. Other internal variables derive their contents from this value.

Related Information:

- [File Names](#)

-----

## SrcDIR

This variable is derived from [SourceNAME](#) and reflects any drive or path information contained therein. For instance, if the value of [SourceNAME](#) is "D:\Source\Dump\DumpMain.asm", then the value of **SrcDIR** would be "D:\Source\Dump\". If no drive or path information was specified in the file name, then **SrcDIR** will contain the empty string.

Related Information:

- [File Names](#)

-----

## SrcEXT

This variable contains the default source file name extension that is conditionally appended to the concatenated values of [SrcDIR](#) and [SrcNAME](#); the assembler treats the resulting string as the fully qualified input file name. The default value for **SrcEXT** is ".asm" unless altered by use of the **Fes** parameterized command line option.

Related Information:

- [Options](#)
- [File Names](#)
- [Fes - Control Source File Extension \(SrcEXT\)](#)

-----

## SrcNAME

This variable contains the "root file name" portion of the source file name, which is extracted from the contents of the [SourceNAME](#) variable. For instance, if the value of [SourceNAME](#) is "D:\Source\Dump\DumpMain.asm", then the value of **SrcNAME** would be "DumpMain". **SrcNAME** should never contain the empty string unless the input file name was incorrectly specified; in which case the assembler will generate an error when it tries to access the file.

Related Information:

- [File Names](#)

-----

## Using ALP

This chapter tells you how to:

1. Invoke and use ALP
2. Use environment variables to pass information to ALP

---

## Invoking ALP

To invoke ALP from the command line, type:

```
alp
```

You can also invoke ALP by absolute reference from a makefile or command file; to do this, the directory should be among those referenced by the [PATH](#) environment variable.

---

## Using Environment Variables

This section describes the environment variables that you can set and that are used by ALP.

---

### \_INCLUDE

When ALP processes an **INCLUDE** directive, ALP translates the value of the [BaseEXE](#) internal variable to uppercase and uses this value to construct the name of an ALP-specific environment variable having the form:

```
<BaseEXE>_INCLUDE
```

For example, If the value of [BaseEXE](#) is *alp*, then ALP constructs an environment variable called **ALP\_INCLUDE** and tries to locate it in the environment. If found, its contents would be expected to contain a list of directories in a format identical to that of the standard [INCLUDE](#) environment variable.

---

### \_OPTIONS

ALP translates the value of the [BaseEXE](#) internal variable to uppercase and uses this value to construct the name of an ALP-specific environment variable having the form:

```
<BaseEXE>_OPTIONS
```

For example, if the value of [BaseEXE](#) is *alp*, then ALP constructs an environment variable called **ALP\_OPTIONS** and tries to locate it in the environment. If found, its contents are logically prepended to the assembler command line.

You can use this variable to set alternate default values for assembler command line options. For maximum flexibility, it is recommended that this variable contain a reference to a command line response file using an **@Filename** directive, which allows the default command line options to be stored in a file rather than in the environment variable itself.

---

### \_PATH

Whenever ALP needs to search for one of its own component files (such as the messages file), the value of the [BaseEXE](#) internal variable is translated to uppercase and is used to construct the name of an ALP specific environment variable having the form "<[BaseEXE](#)>\_PATH." For example, if the value of [BaseEXE](#) is "alp," then an environment variable called **ALP\_PATH** would be constructed and an attempt would be made to locate it in the environment. If found, its contents would be expected to contain a list of directories in a format identical to that of the standard [PATH](#) environment variable. ALP then searches this list of paths when attempting to locate the component file.

---

## DPATH

The **DPATH** environment variable may be utilized for the same purposes as the <[BaseEXE](#)>\_PATH internal variable if so desired.

---

## INCLUDE

The **INCLUDE** environment variable may be utilized for the same purposes as the <[BaseEXE](#)>\_INCLUDE internal variable if so desired.

---

## PATH

The **PATH** environment variable may be utilized for the same purposes as the <[BaseEXE](#)>\_PATH internal variable if so desired.

---

## Using the Command Line

This section describes command line parameter types, syntax, and options.

---

## Command Line Parameter Types

Command line *parameters* are individual "words," or patterns of characters separated by white space. Each individual parameter is recognized by the command line lexical analyzer as having a certain "pattern," and is thus assigned a *parameter type*, as described in the following sections. Parameters should be separated by one or more blanks, tabs, or (when reading from a response file) new line characters, and double quotation marks may be used on the command line to remove the special meaning from the operating system metacharacters. Although the host operating system may support the enclosing of command line parameters within double quotes (") (known as "quoting"), the ALP command line parser also performs quote interpretation. This is necessary to properly interpret quoted parameters within **@Filename** response files, for which there is no built-in support provided by the default operating system command shell.

Parameter types are determined by looking at the first character of each individual "word." Options begin with a plus (+) or minus (-), and file names begin with any other legal file name character (as dictated by the operating system). A special case is a word beginning with the *at sign* (@) character, which signifies the beginning of the **@Filename** (read from a response file) directive.

---

## Options

Options appear on the command line as mnemonic identifiers prefixed by either of the plus (+) or minus (-) characters, and must be separated from other command line parameters by at least one blank character. Case is not significant in option identifiers.

A single option may be specified more than once on the command line within a given scope; the last occurrence overrides all previous definitions within that scope unless the effect of the option is to collect information in a cumulative fashion. Options are not cumulative unless

documented otherwise on an individual basis.

There are two forms of options:

- [Switch Option](#)
- [Parameterized Option](#)

Some options may actually combine both functions of the *switched* and *parameterized* variations; for instance, the **+FI** switch option "turns on" the creation of a listing file, while a parameterized option of the same name (for example, **+FI:george.lst**) has the same effect, but also treats the argument field as the name of the listing file to create.

---

## Switch Option

*Switch Options* represent a Boolean value (**on** or **off**, **yes** or **no**, **true** or **false**) for the identifier specified in the option. The plus (+) or minus (-) character introducing the option specifies the value of the switch; "+" is equivalent to **on**, **yes**, or **true**; and "-" is equivalent to **off**, **no**, or **false**.

Because plus (+) is not a character traditionally used to introduce a command line option, ALP provides an alternate method of specifying a switch option that resembles a more commonly used syntax. The character that affects the actual value of the "switch" (that is, the (+) or (-) character) may also be specified directly after the option identifier; in this case the option must still be introduced by either the (+) or (-) character, but the trailing "switch value" takes precedence.

The following are examples of *Switch Options*:

```
+ML
-m1+
+F1
-F1-
```

---

## Parameterized Option

*Parameterized Options* are introduced in the same manner as switch options, but are instead followed by a colon (:) or an equals sign (=) (with no intervening blank space) to indicate that the option takes one or more *arguments*. The format of the argument field is option specific.

Using the plus (+) character versus the minus (-) character to introduce a parameterized option may or may not have an effect upon how the option is interpreted. Refer to the description of each individual option for details.

The following are examples of *Parameterized Options*:

```
-F1=Zappa.lst
-Sv:M510
+fo="\obj\dd\driver.obj"
-m:127-
```

---

## File Names

A file name may be used as an argument to certain command line options or as a stand-alone command line parameter. The file name character set and naming conventions are operating system dependent, and are treated as transparently as possible by ALP. The use of operating system metacharacters in file names should be avoided, and file names should not begin with the plus (+) or minus (-) characters.

Any file name may be "qualified" with drive or path information as appropriate for the host operating system. ALP accepts both the forward slash (/) and the backward slash (\) as legal path name characters, as well as the colon (:) character. Care must be exercised however, because the underlying operating system may reject the usage of some of these characters.

-----

Global Options      Group Options

The OS/2 version of the assembler is enabled to accept *wild-card characters* (?) and (\*) in file names, which emulates the UNIX ability to expand a single file name specification into a list of all files that match the wild-card pattern. The ? character matches any single file name character in the given position, and the \* character matches any number of file name characters.

-----

---

- Requesting a listing file be generated for all files within the group
- Specifying the target directory for all generated object files
- Controlling the display of warning and informational messages for all files within the group

At any point on the command line, a new *scope* may be opened using the scope operator (`()`). The scope operator effectively creates a new

logical command line whose contents are enclosed in parentheses and is parsed in isolation from other scopes. Any group options in effect at the time the new scope is opened are inherited and applied to all files named within.

---

## Command Line Options

This section describes all the ALP command line options. For each option, a table appears in the description section with the following format:

```
Type Global Group File Default
... ..
```

The values appearing in this table are defined as follows:

Type	This field specifies the <i>type</i> of the option described in that row, and can be one of:  <b>S</b> - <a href="#">Switch Option</a> <b>P</b> - <a href="#">Parameterized Option</a>
Global	Specifies whether or not the option is valid only in a <i>global</i> context; that is, in the outermost scope on the command line. These options typically have meaning only for the assembler executable itself, and not for any files to be processed.
Group	Specifies whether or not the option is valid in a <i>group</i> context; that is, if the option may be applied to multiple files within a given scope without causing ambiguities.
File	Specifies whether or not the option is valid only in a <i>file</i> context; that is, if the option may only be applied to a single file within a given scope.
Default	This field shows the default value for the option being described.

---

## Base Options

This section describes the standalone base options that are defined by a single unique mnemonic identifier character.

---

### D - Define Text Macro

This option allows the definition of a symbolic identifier that becomes visible during the assembly of the input file. A single parameter must be specified using one of the following forms:

```
Name[=Value]
```

```
Name[:Value]
```

The *Name* entry must have the same lexical syntax as a normal assembler [Identifier](#). The *Name* entry is converted to a [Text-EquateName](#) before assembly begins. If no explicit value is specified for the name, then it is assigned the empty string. This is equivalent to specifying the following assembler statement:

```
Name EQU <>
```

If an explicit value is to be assigned, the *Name* entry must be immediately followed by a colon (:) or equals sign (=) delimiter with no intervening spaces. Blank characters may be specified between the delimiter and the value field. The *Value* field may contain any text data, but it must be enclosed in double quotes (") if it contains blanks, tabs, or operating system metacharacters (such as & or |).

**Note:** If quotes are used to specify a value containing embedded blanks or tabs, then at least one blank is required between the delimiter (colon or equals sign) and the opening quote of the value field. For example:

```
-D:NAME= "This string will be correctly interpreted"
-D:NAME="This will not; no blank after the equals sign"
```

Type	Global	Group	File	Default
P	Yes	Yes	Yes	(no default value)

# I - Specify Include File Search Path

See [Fdi - Specify Include File Search Path](#).

## File Control Options

All options that perform file or file name manipulation are described in this section. File Control Options begin with the letter "F", and the last letter of the option identifier specifies the type of file or file name to which the option applies as follows:

<b>d</b>	Dependency File
<b>i</b>	Include File
<b>l</b>	Listing File
<b>m</b>	Messages File
<b>o</b>	Object File
<b>s</b>	Source File

# Fd - Produce Make Dependency File

Turn this flag on to produce a Make dependency file. Using the parameterized version of the option allows the dependency file to be explicitly named.

The dependency file is written in the format understood by all standard "Make" utilities. It contains the name of the target object file, followed by a list of any source files read during assembly by way of the INCLUDE preprocessor directive. The primary source file is NOT listed in this file as a dependent of the object module.

The dependency file is in a format that can be included directly by most Make utilities, but some post-processing of the file may be required for certain build environments.

Type	Global	Group	File	Default
S	Yes	Yes	Yes	-Fd (no dependency file is generated)
P	No	No	Yes	-Fd:<DepDIR><DepNAME>[<DepEXT>] (A dependency file name is generated using the values of the referenced internal variables. The <DepEXT> extension is appended if this feature is turned on.)

---

## Fl - Produce Listing File

Turn this flag on to produce an assembler listing file. Using the parameterized version of the option allows the listing file to be explicitly named.

Only this option controls the actual creation of a listing file; [Listing Control Options](#) have no effect if this option has not been turned on.

Type Global Group File Default

S	Yes	Yes	Yes	-Fl (no listing file is generated)
P	No	No	Yes	-Fl:<LstDIR><LstNAME>[<LstEXT>] (A listing file name is generated using the values of the referenced internal variables. The <a href="#">LstEXT</a> extension is appended if this feature is turned on.)

---

## Fm - Produce Messages File

Turn this flag on to produce a messages file. Using the parameterized version of the option allows the messages file to be explicitly named.

Within the context of a given assembly, by default all error, warning, and informational messages are printed to the standard output device. Use of the **Fm** option allows these messages to be redirected to a separate file; this can be useful when dissecting the output from multiple assemblies. Messages with a severity greater than **Error** are printed to the standard error device, and do not appear in the messages file.

Type Global Group File Default

S	Yes	Yes	Yes	-Fm (no messages file is generated; all messages are printed on the standard output)
P	No	No	Yes	-Fm:<MsgDIR><MsgNAME>[<MsgEXT>] (A messages file name is generated using the values of the referenced internal variables. The <a href="#">MsgEXT</a> extension is appended if this feature is turned on.)

---

## Fo - Produce Object File

An object file name is generated using the values of the referenced internal variables. The [ObjEXT](#) extension is appended if this feature is turned on.

By default, this switch is turned on and thus an object file is produced (provided the assembly completes without errors); this switch may be turned off if an object file is not desired. Using the parameterized version of the option allows the object file to be explicitly named.

Type	Global	Group	File	Default	
S	Yes	Yes	Yes	+Fo	(an object file is generated)
P	No	No	Yes	-Fo:	<ObjDIR><ObjNAME>[<ObjEXT>]

## Fdd - Directory to Store Make Dependency File (DepDIR)

This option affects the [DepDIR](#) variable and allows the user to specify a target directory where the Make dependency file(s) will be stored; by default this variable is empty and dependency file(s) are created in the current working directory. This value is ignored if the **Fd** option was used to explicitly name the dependency file, and the name included absolute or relative path information.

If the value specified in this option is anything other than an unadorned *drive letter* (for example, **D:**) or a string ending with a *path separator character* (**/** or **\**), then the *path separator character* appropriate for the underlying operating system is appended to the string.

Type	Global	Group	File	Default	
P	Yes	Yes	Yes	-Fdd:	<DepDIR>

## Fdi - Specify Include File Search Path

This option accepts a path (or list of paths separated by semicolons) that are searched by the assembler while attempting to locate an **INCLUDE** file. When multiple occurrences of this option are specified within a given scope, the effect is cumulative rather than destructive; successive occurrences add to the existing list rather than overwriting previous definitions. The more conventional spelling "**I**" can be used as an alias for the **Fdi** option.

Type	Global	Group	File	Default	
P	Yes	Yes	Yes	-Fdi:	<IncDIR>

## Fdl - Directory to Store Listing File (LstDIR)

This option affects the [LstDIR](#) variable and allows the user to specify a target directory where the listing file(s) will be stored; by default this variable is empty and listing file(s) are created in the current working directory. This value is ignored if the **Fl** option was used to explicitly name the listing file, and the name included absolute or relative path information.

If the value specified in this option is anything other than an unadorned *drive letter* (for example, **D:**) or a string ending with a *path separator character* (**/** or **\**), then the *path separator character* appropriate for the underlying operating system is appended to the string.

Type	Global	Group	File	Default	
P	Yes	Yes	Yes	-Fdl:	<LstDIR>

---

## Fdm - Directory to Store Messages File (MsgDIR)

This option affects the [MsgDIR](#) variable and allows the user to specify a target directory where the messages file(s) will be stored; by default this variable is empty and messages file(s) are created in the current working directory. This value is ignored if the **Fm** option was used to explicitly name the message file, and the name included absolute or relative path information.

If the value specified in this option is anything other than an unadorned *drive letter* (for example, **D:**) or a string ending with a *path separator character* (**/** or **\**), then the *path separator character* appropriate for the underlying operating system is appended to the string.

```
Type Global Group File Default
P      Yes      Yes      Yes  -Fdm:<MsgDIR>
```

---

## Fdo - Directory to Store Object File (ObjDIR)

This option affects the [ObjDIR](#) variable and allows the user to specify a target directory where the object file(s) will be stored; by default this variable is empty and object file(s) are created in the current working directory. This value is ignored if the **Fo** option was used to explicitly name the object file, and the name included absolute or relative path information.

If the value specified in this option is anything other than an unadorned *drive letter* (for example, **D:**) or a string ending with a *path separator character* (**/** or **\**), then the *path separator character* appropriate for the underlying operating system is appended to the string.

```
Type Global Group File Default
P      Yes      Yes      Yes  -Fdo:<ObjDIR>
```

---

## Fds - Directory to Locate Source File (SrcDIR)

This option affects the [SrcDIR](#) variable and allows the user to specify a source directory from which source file(s) will be loaded; by default this variable is empty and source file(s) are searched for in the current working directory. This value is ignored if the source file name included absolute or relative path information.

If the value specified in this option is anything other than an unadorned *drive letter* (for example, **D:**) or a string ending with a *path separator character* (**/** or **\**), then the *path separator character* appropriate for the underlying operating system is appended to the string.

```
Type Global Group File Default
P      Yes      Yes      Yes  -Fds:<SrcDIR>
```

---

# Fed - Control Make Dependency File Extension (DepEXT)

This option determines whether or not the value of the [DepEXT](#) variable is appended to Make dependency file names. The parameterized version of this option affects the actual value of the [DepEXT](#) variable.

Type	Global	Group	File	Default
S	Yes	Yes	Yes	-Fed (the value of <a href="#">DepEXT</a> is not appended to dependency file names)
P	Yes	Yes	Yes	-Fed:< <a href="#">DepEXT</a> >

---

# Fei - Control Include File Extension (IncEXT)

This option determines whether or not the value of the [IncEXT](#) variable is appended to file names generated by the preprocessor when processing the INCLUDE directive. The parameterized version of this option affects the actual value of the [IncEXT](#) variable.

Type	Global	Group	File	Default
S	Yes	Yes	Yes	-Fei (the value of <a href="#">IncEXT</a> is not appended to include file names)
P	Yes	Yes	Yes	-Fei:< <a href="#">IncEXT</a> >

---

# Fel - Control Listing File Extension (LstEXT)

This option determines whether or not the value of the [LstEXT](#) variable is appended to listing file names. The parameterized version of this option affects the actual value of the [LstEXT](#) variable.

Type	Global	Group	File	Default
S	Yes	Yes	Yes	+Fel (the value of <a href="#">LstEXT</a> is appended to listing file names)
P	Yes	Yes	Yes	+Fel:< <a href="#">LstEXT</a> >

---

# Fem - Control Messages File Extension (MsgEXT)

This option determines whether or not the value of the [MsgEXT](#) variable is appended to messages file names. The parameterized version of

this option affects the actual value of the [MsgEXT](#) variable.

Type	Global	Group	File	Default
S	Yes	Yes	Yes	+Fem (the value of <a href="#">MsgEXT</a> is appended to messages file names)
P	Yes	Yes	Yes	+Fem:< <a href="#">MsgEXT</a> >

-----

## Feo - Control Object File Extension (ObjEXT)

This option determines whether or not the value of the [ObjEXT](#) variable is appended to object file names. The parameterized version of this option affects the actual value of the [ObjEXT](#) variable.

Type	Global	Group	File	Default
S	Yes	Yes	Yes	+Feo (the value of <a href="#">ObjEXT</a> is appended to object file names)
P	Yes	Yes	Yes	+Feo:< <a href="#">ObjEXT</a> >

-----

## Fes - Control Source File Extension (SrcEXT)

This option determines whether or not the value of the [SrcEXT](#) variable is appended to source file names. The parameterized version of this option affects the actual value of the [SrcEXT](#) variable.

Type	Global	Group	File	Default
S	Yes	Yes	Yes	+Fes (the value of <a href="#">SrcEXT</a> is appended to source file names)
P	Yes	Yes	Yes	+Fes:< <a href="#">SrcEXT</a> >

-----

## Listing Control Options

This section describes all options related to controlling the content of the assembler listing file. All listing control options begin with the letter "L".

Options that manipulate the characteristics of individual listing file columns reference a particular column by having a single character mnemonic identifier as part of the option identifier. Listing column mnemonics are as follows:

<b>C</b>	<b>Conditional assembly nesting level</b> is a numeric value that appears during processing of a conditional assembly directive and is incremented for each level of nesting that occurs.
<b>D</b>	<b>Macro definition line number</b> tracks line numbers for each new MACRO definition introduced into the assembly.
<b>F</b>	<b>True or false conditional flag</b> appears during processing of a conditional assembly directive and is either a plus (+) character to denote that the conditional expression was TRUE and tokens appearing within the block are being interpreted, or a minus (-) character to denote that the conditional expression was FALSE and tokens appearing within the block are being ignored.
<b>G</b>	<b>Generated machine code data</b> column shows the hexadecimal values for data generated by machine instructions or data allocation statements.
<b>I</b>	<b>Include file nesting level</b> is a numeric value that appears during processing of INCLUDE files and is incremented for each level of nesting that occurs.
<b>L</b>	<b>Macro expansion indentation level</b> is a text field whose width reflects the current nesting level of expanded macros, and whose value contains a simulated "arrow" using the "--->" characters.
<b>M</b>	<b>Macro expansion nesting level</b> is a numeric value that appears during macro expansions and is incremented for each level of nesting that occurs.
<b>O</b>	<b>Location counter offset value</b> is a numeric value displayed in hexadecimal notation and indicates the current offset of the location counter within the current segment or structure.
<b>S</b>	<b>Source line data</b> column contains the text data of the current line in the input source file.
<b>X</b>	<b>Cumulative listing line number</b> is incremented for every new line that appears in the listing file.
<b>Y</b>	<b>Individual source file line number</b> tracks line numbers for the top-level source file and for each separate INCLUDE file.
<b>Z</b>	<b>Macro expansion line number</b> tracks the current line number for each MACRO expanded during the assembly.

-----

## Lc\* - Control Display of Individual Columns

This family of options controls whether or not an individual column physically appears in the listing file. The display of each column may be controlled with a switch option by using the standard ON (+) or OFF (-) switch values (see [Switch Option](#)) or by using the parameterized option syntax (see [Parameterized Option](#)) with one of the following keyword values in the argument field:

<b>B</b>	Abbreviation for BLANK.
<b>BLANK</b>	The column will appear as a place-holder in the listing file, but the column data will not be displayed.
<b>OFF</b>	The column will not be displayed.
<b>ON</b>	The column will be displayed.
<b>Z</b>	Abbreviation for ZBLANK.
<b>ZBLANK</b>	The column data will only display if its value is nonzero (valid only for numeric fields).

Type Global Group File Default

S	Yes	Yes	Yes	+LcX (display Cumulative Listing Line Number)
S	Yes	Yes	Yes	+LcY (display Individual Source File Line Number)
S	Yes	Yes	Yes	-LcZ:Z (display Macro Expansion Line Number if not zero)
S	Yes	Yes	Yes	-LcD:Z (display Macro Definition Line Number if not zero)
S	Yes	Yes	Yes	+LcL (display Macro Expansion Indentation Level)
S	Yes	Yes	Yes	-LcM:Z (display Macro Expansion Nesting Level if not zero)
S	Yes	Yes	Yes	-LcI:Z (display Include File Nesting Level if not zero)
S	Yes	Yes	Yes	+LcC (display Conditional Assembly Nesting Level)

S	Yes	Yes	Yes	+LcF (display True or False Conditional Flag)
S	Yes	Yes	Yes	+LcO (display Location Counter Offset Value)
S	Yes	Yes	Yes	+LcG (display Generated Machine Code Data)
S	Yes	Yes	Yes	+LcS (display Source Line Data)

-----

## Lcm\* - Specify Left Margin for Individual Columns

This family of options specifies the left margin value for each individual column, which determines the number of blank spaces that will appear to the left of the column data.

Type	Global	Group	File	Default
P	Yes	Yes	Yes	-LcmX:0 (Cumulative Listing Line Number)
P	Yes	Yes	Yes	-LcmY:1 (Individual Source File Line Number)
P	Yes	Yes	Yes	-LcmZ:1 (Macro Expansion Line Number)
P	Yes	Yes	Yes	-LcmD:1 (Macro Definition Line Number)
P	Yes	Yes	Yes	-LcmL:0 (Macro Expansion Indentation Level)
P	Yes	Yes	Yes	-LcmM:1 (Macro Expansion Nesting Level)
P	Yes	Yes	Yes	-LcmI:1 (Include File Nesting Level)
P	Yes	Yes	Yes	-LcmC:1 (Conditional Assembly Nesting Level)
P	Yes	Yes	Yes	-LcmF:1 (True or False Conditional Flag)
P	Yes	Yes	Yes	-LcmO:2 (Location Counter Offset Value)
P	Yes	Yes	Yes	-LcmG:2 (Generated Machine Code Data)
P	Yes	Yes	Yes	-LcmS:2 (Source Line Data)

-----

## Lct\* - Specify Truncation of Individual Columns

This family of options specifies whether or not the data contained within an individual column will be truncated if it exceeds the column width, or whether it will overflow onto additional lines until the entire column contents have been printed.

Type	Global	Group	File	Default
S	Yes	Yes	Yes	+LcmX (truncate Cumulative Listing Line Number)
S	Yes	Yes	Yes	+LcmY (truncate Individual Source File Line Number)
S	Yes	Yes	Yes	+LcmZ (truncate Macro Expansion Line Number)
S	Yes	Yes	Yes	-LcmD (do not truncate Macro Definition Line Number)
S	Yes	Yes	Yes	-LcmL (do not truncate Macro Expansion Indentation

				Level)
S	Yes	Yes	Yes	+LcmM (truncate Macro Expansion Nesting Level)
S	Yes	Yes	Yes	+LcmI (truncate Include File Nesting Level)
S	Yes	Yes	Yes	+LcmC (truncate Conditional Assembly Nesting Level)
S	Yes	Yes	Yes	+LcmF (truncate True or False Conditional Flag)
S	Yes	Yes	Yes	-LcmO (do not truncate Location Counter Offset Value)
S	Yes	Yes	Yes	-LcmG (do not truncate Generated Machine Code Data)
S	Yes	Yes	Yes	+LcmS (truncate Source Line Data)

-----

## Lcw\* - Specify Width of Individual Columns

This family of options specifies the width of each individual listing column in single character positions. Note that the width of column **L** (Macro Expansion Indentation Level) will vary according to the macro expansion nesting level (which is also displayed as a numeric value in column **M**) if the nesting level value exceeds the column width. This behavior may be avoided by setting the width of column **L** such that its width never exceeds the value of column **M**, or by turning off the display of column **L** altogether.

Type	Global	Group	File	Default
P	Yes	Yes	Yes	-LcmX:4 (Cumulative Listing Line Number)
P	Yes	Yes	Yes	-LcmY:4 (Individual Source File Line Number)
P	Yes	Yes	Yes	-LcmZ:3 (Macro Expansion Line Number)
P	Yes	Yes	Yes	-LcmD:3 (Macro Definition Line Number)
P	Yes	Yes	Yes	-LcmL:0 (Macro Expansion Indentation Level)
P	Yes	Yes	Yes	-LcmM:2 (Macro Expansion Nesting Level)
P	Yes	Yes	Yes	-LcmI:2 (Include File Nesting Level)
P	Yes	Yes	Yes	-LcmC:2 (Conditional Assembly Nesting Level)
P	Yes	Yes	Yes	-LcmF:1 (True or False Conditional Flag)
P	Yes	Yes	Yes	-LcmO:4 (Location Counter Offset Value)
P	Yes	Yes	Yes	-LcmG:24 (Generated Machine Code Data)
P	Yes	Yes	Yes	-LcmS:90 (Source Line Data)

-----

## Lc - Control display of false Conditional blocks

This switch determines whether or not sections of source code appear in the listing file when they are rendered inactive by a false conditional expression. By default, the assembler does not show source code in the listing file if it is skipped during conditional processing; turn this switch on if listing of all source code is desired.

Type Global Group File Default

S Yes Yes Yes -Lc (do not list false conditional blocks)

---

## Ld - Control Display of Listing Directives

This switch controls whether or not assembler listing directives appear in the listing output. Listing directives are shown by default; turn this switch off to hide them.

Type Global Group File Default

S Yes Yes Yes +Lc (show all listing directives)

---

## Le - Control Display of Error/Warning/Info Messages

By default, any time the assembler prints an Error, Warning, or Info message during the assembly, the message also appears in the listing file following the source line to which it refers. Turn this switch off if such messages are not desired in the listing output.

Type Global Group File Default

S Yes Yes Yes +Le (show messages in listing file)

---

## Lf - Control Use of FormFeed Characters

When the assembler is generating formatted listing output and it needs to advance to the next page, it inserts the ASCII FormFeed character (0x0C) into the listing output stream. If this causes problems, turning this switch off will instead cause the assembler to generate the appropriate number of newline character sequences to perform the page eject operation.

Type Global Group File Default

S Yes Yes Yes +Lf (the FormFeed character is used)

---

## Li - Control Display of INCLUDE Files

When the assembler processes source code stored in an INCLUDE file, by default the contents of the file are expanded in the listing output;

depending on the types of files that are included, this behavior can result in large volumes of listing output. Turn this switch off if the expansion is not desired.

Type	Global	Group	File	Default
------	--------	-------	------	---------

S	Yes	Yes	Yes	+Li (INCLUDE files are expanded in listing output)
---	-----	-----	-----	----------------------------------------------------

-----

## Llp - Specify Length of Page

To correctly format the listing file for subsequent hardcopy output, the assembler must know how many physical lines of output will fit vertically on the printed page. This setting is especially important if the use of FormFeed characters has been turned off with the **Lf** option. The default value for this option is 66 lines per page.

Type	Global	Group	File	Default
------	--------	-------	------	---------

P	Yes	Yes	Yes	-Llp:66 (the default page length is 66 lines)
---	-----	-----	-----	-----------------------------------------------

### Related Information:

- [Lf - Control Use of FormFeed Characters](#)
- [Lwp - Specify Width of Page](#)

-----

## Lm - Control Display of Macro Expansions

This switch controls whether or not the text body of an expanded macro appears in the listing output. While turning this switch on can be very useful when debugging macros, it can also result in large volumes of listing output if many macros are utilized. By default, macro expansions do not appear in the listing output; turn this switch on if this behavior is desired.

Type	Global	Group	File	Default
------	--------	-------	------	---------

S	Yes	Yes	Yes	-Lm (macro expansions do not appear in listing output)
---	-----	-----	-----	--------------------------------------------------------

-----

## Lmb - Specify Bottom Margin

This option determines how many blank lines will appear at the bottom of the page in the listing output; by default this value is 4. The correct behavior of this option depends on the setting of the **Llp** and **Lf** options, and that they match the settings of the physical output device. If there are problems with these settings, then the actual bottom margin may not appear to correctly reflect the value of this option.

Type	Global	Group	File	Default
------	--------	-------	------	---------

P      Yes      Yes      Yes      -Lmb:4 (4 blank lines at bottom of page)

**Related Information:**

- [Lf - Control Use of FormFeed Characters](#)
- [Llp - Specify Length of Page](#)
- [Lmb - Specify Middle Margin after Title](#)
- [Lmt - Specify Top Margin before Title](#)

-----

## Lml - Specify Left Margin

This option specifies the number of blank characters that are printed to the left of every line of listing output. The default value for this option is 4.

Type Global Group File Default

P      Yes      Yes      Yes      -Lml:4 (left margin is 4 blank characters wide)

**Related Information:**

- [Lmr - Specify Right Margin](#)
- [Lwp - Specify Width of Page](#)
- [Lcm\\* - Specify Left Margin for Individual Columns](#)

-----

## Lmb - Specify Middle Margin after Title

This option specifies the number of blank lines that separate the assembler heading, the title and subtitle (if there are any), from both the *column ruler* (if there is one) and the body of the generated listing text. The default value for this option is 2 blank lines.

Type Global Group File Default

P      Yes      Yes      Yes      -Lmm:2 (2 blank lines after title and subtitle,  
and before ruler line)

**Related Information:**

- [Llp - Specify Length of Page](#)
- [Lmb - Specify Bottom Margin](#)
- [Lmt - Specify Top Margin before Title](#)
- [Lr - Control Display of Column Ruler](#)

-----

## Lmr - Specify Right Margin

This option specifies the number of blank characters that are reserved (but not actually printed) to the right of every line of listing output. The default value for this option is 4.

Type Global Group File Default

P Yes Yes Yes -Lmr:4 (right margin is 4 blank characters wide)

#### Related Information:

- [Lml - Specify Left Margin](#)
- [Lwp - Specify Width of Page](#)
- [Lcm\\* - Specify Left Margin for Individual Columns](#)

-----

## Lmt - Specify Top Margin before Title

This option specifies the number of blank lines that appear at the top of the page before any other listing output is generated. The default value for this option is 2 blank lines.

Type Global Group File Default

P Yes Yes Yes -Lmt:2 (2 blank lines at the top of the page)

#### Related Information:

- [Llp - Specify Length of Page](#)
- [Lmb - Specify Bottom Margin](#)
- [Lmb - Specify Middle Margin after Title](#)

-----

## Lp - Generate Listing on Specific Pass

This option allows the user to control whether or not listing information is generated on a specific pass of the assembler. By default, the assembler only generates listing information on pass two. If the user is encountering "phase errors" or other unusual situations, it may be helpful to request a listing for the first pass as well.

The arguments to this option are either a series of numeric digits (without intervening white space) or the **ALL** or **NONE** keywords. In the default assembler configuration, use of the **-Lp:ALL** form is equivalent to specifying **-Lp:12**, because the assembler makes two passes through the source file by default. The **NONE** keyword prevents generation of any pass-related information in the listing file; however, symbol table information will still appear if selected.

When using numeric digits to specify the desired pass numbers, a listing will only be generated for the numbers given in the argument field; the default setting (or settings given by previous occurrences of the option) will be discarded.

Type Global Group File Default

P Yes Yes Yes -Lp:2 (listing on pass 2 only)

-----

## Lr - Control Display of Column Ruler

This switch determines whether or not the *column ruler* appears at the top of each page in the listing output. This ruler is simply a line of information containing a string of alphabetic characters corresponding to each vertical column of listing information. The ruler reflects the current width, margins, and placement of the various listing columns at the time each page is printed, and helps the user to determine which column they are looking at. Turn this switch off if display of the column ruler is not desired.

Type Global Group File Default

S    Yes    Yes    Yes    +Lr (show the column ruler in listing output)

-----

## Ls - Control Display of Symbol Table

This switch determines whether or not a summary of the symbol table contents is included at the end of the listing file. The default behavior is to omit the symbol table summary; turn this switch on to include it.

Type Global Group File Default

S    Yes    Yes    Yes    -Ls (do not include symbol table in listing  
output)

-----

## Lt1 - Specify Title

This option allows the user to specify the text of a default title to be printed at the top of each listing page; there is no default title. Title information must be enclosed in double quotes "" if it contains white space characters.

Type Global Group File Default

P    Yes    Yes    Yes    -Lt1:<empty> (no default title information)

-----

## Lt2 - Specify Subtitle

This option allows the user to specify the text of a default subtitle to be printed at the top of each listing page; there is no default subtitle. Subtitle information must be enclosed in double quotes "" if it contains white space characters.

Type Global Group File Default

P    Yes    Yes    Yes    -Lt2:<empty> (no default subtitle information)

---

## Lwp - Specify Width of Page

To correctly format the listing file for subsequent hardcopy output, the assembler must know how many physical characters of output will fit horizontally on the printed page. The default value for this option is 132 character positions.

Type Global Group File Default

P	Yes	Yes	Yes	-Lwp:132 (the default page width is 132 character positions)
---	-----	-----	-----	--------------------------------------------------------------

### Related Information:

- [Lcm\\* - Specify Left Margin for Individual Columns](#)
- [Lcw\\* - Specify Width of Individual Columns](#)

---

## Lwt - Specify Tab Expansion Width

This option specifies the width of a tab character in blank spaces. Tab characters appearing in the source file are always expanded into blank spaces when output to the listing file; the default behavior is to expand tab characters to every eighth character position.

Type Global Group File Default

P	Yes	Yes	Yes	-Lwt:8 (tab characters are 8 character positions wide)
---	-----	-----	-----	--------------------------------------------------------

---

## Message Control Options

This section describes all options related to the output and control of assembler messages. All Message Control Options begin with the letter "M".

---

## M - Control Individual Messages or Groups

This option controls the types of messages that are displayed by manipulating *message group identifier flags* or individual message numbers. Only messages with a severity of **Warning** or **Info** are controllable with this option. Messages with a severity of **Error**, **System**, **Fatal**, **Internal**, or **Usage** cannot be suppressed.

Type Global Group File Default

P	Yes	Yes	Yes	-M:W+ (all warning messages are enabled)
---	-----	-----	-----	------------------------------------------

All assembler messages are assigned a unique message number, and **Warning** or **Info** messages may belong to one or more message groups. The message group identifier flags are defined as follows:

<b>ALL</b>	All warning and informational messages
<b>BLK</b>	Messages regarding block structure violations
<b>COD</b>	Messages regarding code generation
<b>FIL</b>	File manipulation messages
<b>I</b>	All informational messages
<b>PP</b>	Preprocessor messages
<b>SRC</b>	Source file lexical analyzer messages
<b>STA</b>	Assembly statistics
<b>W</b>	All warning messages

Any sequence of message groups or message numbers may be specified in the argument field of the **M** option; each argument must be followed by a plus (+) or minus (-) character to turn the value on or off, and no intervening white space characters may appear between arguments.

See [Assembler Messages](#) for more information on message number values and the messages groups to which they belong.

-----

## Mb - Control Printing of the Assembler Banner

This switch controls whether or not the assembler start-up banner is printed. This switch is on by default; turn it off to suppress display of the banner.

```
Type Global Group File Default
S      Yes      No      No      +Mb (Print the assembler banner)
```

-----

## Me - Set Number of Errors Before Assembler Aborts

This option specifies the maximum number of errors that the assembler will tolerate before ending the assembly. The default value is 50.

```
Type Global Group File Default
P      Yes      Yes      Yes      -Me:50 (abort the assembly after 50 errors are
                                     encountered)
```

-----

## Mwe - Treat Warnings as Errors

This switch tells the assembler that any Warning messages are to be treated as though they were errors; this causes the assembler to end with a non-zero exit code, and helps prevent any warning conditions from "passing by" unnoticed.

```
Type Global Group File Default
S      Yes      Yes      Yes      -Mwe (warnings are not considered to be errors)
```

---

# Object Control Options

This section describes all options related to the output and control of object file information. All Object Control Options begin with the letter "O".

---

## Od - Line Number and Symbolic Debug Information in Object File

This switch controls whether or not all forms of debug information are included in the object file, and is a shorthand method of specifying the options to control *line numbering* and *symbolic* debug information.

The parameterized version of this option may be used to specify the format of the debugging information. The setting of this value may be necessary depending on which linker is used to link the object file output, or on the debugger used to debug the executable. The argument must be one of the following keywords:

**IBM32**           Generate debugging information in the 32-bit IBM (HLL) format. Executable code that is to be debugged with the IBM family of debuggers should use this setting. This is the default value if no parameter is specified.

**MS16**           Generate debugging information in the 16-bit Microsoft (CodeView) format. You may need to use this setting if the object file output will be processed by a 16-bit linker, or if non-IBM debuggers will be utilized on the executable.

Type	Global	Group	File	Default
------	--------	-------	------	---------

S	Yes	Yes	Yes	(see default values for Ods and Od1)
---	-----	-----	-----	--------------------------------------

P	Yes	Yes	Yes	-Od:IBM32 (use the IBM debug format)
---	-----	-----	-----	--------------------------------------

### Related Information:

- [Od1 - Line Numbering Information in Object File](#)
- [Ods - Symbolic Debug Information in Object File](#)

---

## Od1 - Line Numbering Information in Object File

This switch controls whether or not line numbering debug information is included in the object file, thus allowing the assembler source file to be viewed from within a source-level debugger.

Type	Global	Group	File	Default
------	--------	-------	------	---------

S	Yes	Yes	Yes	-Od1 (line numbering debug information is not included in object file)
---	-----	-----	-----	------------------------------------------------------------------------

---

## Ods - Symbolic Debug Information in Object File

This switch controls whether or not symbolic debug information is included in the object file, thus allowing variables, labels, and expressions appearing in the assembler source file to be viewed from within a source-level debugger.

Type	Global	Group	File	Default
------	--------	-------	------	---------

S	Yes	Yes	Yes	-Ods (symbolic debug information is not included in object file)
---	-----	-----	-----	------------------------------------------------------------------

-----

## Oug - Convert Global Identifiers to Uppercase

This switch controls whether external identifiers (declared with the **EXTERN** directive) and public identifiers (declared with the **PUBLIC** directive) are converted to uppercase before writing them to the object file. By default, no conversion is performed and the symbols are written to the object file exactly as they were entered into the symbol table during assembly.

Type	Global	Group	File	Default
------	--------	-------	------	---------

S	Yes	Yes	Yes	-Oug (do not convert global identifiers to uppercase in object file)
---	-----	-----	-----	----------------------------------------------------------------------

### Related Information:

- [Scs - Control Case Sensitivity for Symbol Names](#)
- [Ous - Convert Group and Segment Names to Uppercase](#)

-----

## Ous - Convert Group and Segment Names to Uppercase

This switch controls whether external group names (declared with the **GROUP** directive) and segment names (declared with the **SEGMENT** directive) are converted to uppercase before writing them to the object file. By default, no conversion is performed and the names are written to the object file exactly as they were entered into the symbol table during assembly.

Type	Global	Group	File	Default
------	--------	-------	------	---------

S	Yes	Yes	Yes	-Ous (do not convert group or segment names to uppercase in object file)
---	-----	-----	-----	--------------------------------------------------------------------------

### Related Information:

- [Scs - Control Case Sensitivity for Symbol Names](#)
- [Oug - Convert Global Identifiers to Uppercase](#)

-----

## Source Control Options

All options related to parsing or processing the input source stream are described in this section. All Source Control Options begin with the

letter "S".

---

## Sc - Control Case Sensitivity for All Identifiers

This switch controls whether or not all identifiers are case sensitive, and is a shorthand method of specifying the options for user identifiers and keywords.

Type	Global	Group	File	Default
S	Yes	Yes	Yes	(see default values for Sck and Scs)

### Related Information:

- [Sck - Control Case Sensitivity for Keywords](#)
- [Scs - Control Case Sensitivity for Symbol Names](#)

---

## Sck - Control Case Sensitivity for Keywords

This switch controls whether or not language keywords are case sensitive. By default, this flag is turned off; thus the spellings **SEGMENT**, **Segment**, and **segment** all refer to the same keyword. Turning this switch on would render the three spellings separate and distinct, and only the uppercase variant would be recognized as a keyword.

This option has no effect on user identifiers (see [Scs - Control Case Sensitivity for Symbol Names](#)) or processor mnemonics.

Type	Global	Group	File	Default
S	Yes	Yes	Yes	-Sck (All language keywords are case insensitive)

---

## Scs - Control Case Sensitivity for Symbol Names

This switch controls whether or not user identifiers are case sensitive. By default, this flag is turned on; thus the identifiers **GEORGE**, **George**, and **george** are separate and distinct. Turning this switch off would cause the three spellings to refer to the same identifier.

If a case-insensitive assembly is being performed (-Scs), the actual spelling of the identifier is not altered (e.g., converted to uppercase) when it is entered into the symbol table. The actual symbol definition controls the spelling of the identifier regardless of whether or not a case-insensitive assembly was performed. The only exception to this rule is when processing a **PUBLIC** directive under MASM 5.10 emulation; the identifier spelling, which appears in the **PUBLIC** directive is honored over the one appearing in the actual identifier definition.

This option has no effect on language keywords (see [Sck - Control Case Sensitivity for Keywords](#)), processor mnemonics, or register names.

Type	Global	Group	File	Default
S	Yes	Yes	Yes	+Scs (All user identifiers are case sensitive)

#### Related Information:

- [Oug - Convert Global Identifiers to Uppercase](#)
- [Ous - Convert Group and Segment Names to Uppercase](#)

---

## Sk - Control Use of Reserved Words as Labels

This switch controls whether or not certain assembler keywords (reserved words) may be used in the context of a code label (for example, **TEST**:). By default, this switch is off, and keywords may not be used as labels.

Even when this switch is turned on, there are severe restrictions on this capability. Processor mnemonics classify as the only "keywords" allowed in this situation, and only in the context of a *code label* (a label followed by a colon); using any reserved word as a directive name or data label is illegal.

Type Global Group File Default

S     Yes     Yes     Yes     -Sk (Reserved words may not be used as labels)

---

## Sfs - SHORT is Default Distance for Forward-Referenced Jumps

By default, when the assembler encounters an unqualified forward reference as the operand to a jump instruction, it makes a worst-case assumption that the target will not be close enough to allow generating the **SHORT** variation of the instruction. Enough space is reserved to generate the **NEAR** version, and if it is determined later that the target is close enough, the **SHORT** variation is generated and extra space is padded with **NOP** instructions. This helps insure that source files will assemble without "out of range" errors, but wastes space when the **NOP** instructions are generated.

Turning this switch on causes the assembler to assume that unqualified forward referenced jumps will always be reachable with the **SHORT** instruction variation; should this not be the case, an error is generated and the user may recode the instruction using the **NEAR** override.

Type Global Group File Default

S     Yes     Yes     Yes     -Sfs (default distance is NEAR)

---

## Sme - Control Visibility of MASM 6.00 Extended Mnemonics

This option controls whether or not the following processor mnemonics are recognized as keywords:

FLDENVD  
FLDENVW  
FNSAVED  
FNSAVEW  
FNSTENV  
FNSTENVW  
FRSTORD  
FRSTORW  
FSAVED  
FSAVEW

FSTENV  
FSTENVW  
IRETF  
IRETDF  
LOOPD  
LOOPW  
LOOPED  
LOOPEW  
LOOPNED  
LOOPNEW  
LOOPNZD  
LOOPNZW  
LOOPZD  
LOOPZW  
POPD  
POPW  
PUSHD  
PUSHW

These mnemonics were introduced in MASM 6.00 to allow explicit word (16-bit) or double-word (32-bit) operations on 80386 or newer processors. Although MASM 5.10 supports the 80386 processor, it does not recognize these keywords. To avoid conflicts with preexisting macro libraries, ALP will also refuse to recognize these keywords when operating under MASM 5.10 compatibility mode (-Sv:M510) unless this option is turned on. This option is turned on automatically when the -Sv:M600 option (or greater) is used.

Type Global Group File Default

S Yes Yes Yes -Sme (extended mnemonics are not enabled)

#### Related Information:

- [Sv - Set Version Behavior](#)

---

## Sv - Set Version Behavior

This option controls the various modes of compatibility that the assembler is designed to emulate. The argument to the **Sv** option must be one of the following keywords:

<b>ALP</b>	Native operating mode; don't emulate other assemblers
<b>M510</b>	Emulate Microsoft MASM Version 5.10
<b>M600</b>	Emulate Microsoft MASM Version 6.00
<b>M611</b>	Emulate Microsoft MASM Version 6.11
<b>M612</b>	Emulate Microsoft MASM Version 6.12
<b>M613</b>	Emulate Microsoft MASM Version 6.13
<b>M614</b>	Emulate Microsoft MASM Version 6.14

Type Global Group File Default

P Yes Yes Yes +Sv:ALP (do not emulate other assemblers)

#### Related Information:

- [Sme - Control Visibility of MASM 6.00 Extended Mnemonics](#)

---

## The MASM2ALP Utility

MASM2ALP is a utility that accepts a MASM 5.10-compatible command line, transforms the command line parameters to the appropriate ALP

syntax, and invokes **alp.exe** to perform the assembly. This allows the use of ALP instead of MASM in existing build environments without requiring major changes to makefiles or build scripts. You simply replace your existing MASM executable with MASM2ALP and resume operations.

MASM2ALP accepts the following MASM 5.10-style command line syntax:

```
masm2alp [Options] SourceFile [, [ObjectFile] [, [ListingFile] [, [CrossRefFile]]] [;]
```

#### Notes:

- **Options** are normally specified first on the command line, but may appear anywhere before the optional semicolon (;) terminator. Options may begin with either a slash (/) or a dash (-), but once the first option is encountered, no further mixing of introductory characters is allowed. Options are not case sensitive. Individual command line options are discussed below.
- Filename arguments are position-dependent. Commas (,) must be used to separate each argument and to maintain argument position, both when arguments are explicitly specified or when they are skipped (left unspecified). The filename argument descriptions are as follows:

##### ***SourceFile***

This argument is mandatory, and specifies the name of the assembly language source file to be processed. If no filename suffix is supplied, a default value of ".asm" is assumed.

##### ***ObjectFile***

The optional name of the object-code output file to be produced. If no filename suffix is supplied, a default value of ".obj" is assumed. If no ***ObjectFile*** argument is given, the root portion of the ***SourceFile*** argument is used as the name of the object file.

##### ***ListingFile***

The optional name of the listing output file to be produced. If no filename suffix is supplied, a default value of ".lst" is assumed. If no ***ListingFile*** argument or **-L** option is given, then no listing file is produced; otherwise the root portion of the ***SourceFile*** argument is used as the name of the listing file.

##### ***CrossRefFile***

The optional name of the cross-reference output file. Since ALP does not support the creation of a cross-reference file, MASM2ALP ignores this argument.

- The trailing semicolon (;) operator may be used when no more filename arguments are to be specified and the default values are to be utilized for the remaining arguments. Any arguments or command line options that follow the semicolon will be ignored.
- MASM2ALP does not support the "prompting" behavior of Microsoft MASM when insufficient command line arguments are supplied. A syntax error is issued in this event.

#### Options:

MASM2ALP accepts the following command line options:

- |                                 |                                                                                                                                                                                                                             |
|---------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>-A</b>                       | Forces MASM to write segments to the object file in alphabetical order. MASM normally writes segments in source code order. MASM2ALP ignores this option.                                                                   |
| <b>-B</b> <i>number</i>         | Sets the size of the source file read buffer in 1K increments. MASM2ALP ignores this option.                                                                                                                                |
| <b>-C</b>                       | Tells MASM to create a cross-reference file. MASM2ALP ignores this option.                                                                                                                                                  |
| <b>-d</b>                       | Causes the assembler to generate listing file information during both pass 1 and 2. By default, the assembler only generates listing file information during pass 2.                                                        |
| <b>-D</b> <i>symbol[=value]</i> | Defines a symbol visible during assembly. An optional text value may also be specified for the symbol.                                                                                                                      |
| <b>-E</b>                       | Tells MASM to generate code compatible with floating-point emulation libraries. MASM2ALP ignores this option.                                                                                                               |
| <b>-H</b>                       | Prints the help information panel for the command line syntax.                                                                                                                                                              |
| <b>-I</b> <i>path</i>           | Specifies an INCLUDE file search path.                                                                                                                                                                                      |
| <b>-L</b> <b>[A]</b>            | Forces the creation of a listing output file. The <b>-LA</b> option is an abbreviation for "list all", which forces macro expansions and false conditionals to also be included in the listing output.                      |
| <b>-ML</b>                      | Directs the assembler to perform a case-sensitive assembly, and to preserve the case of external identifiers when writing them to the object file.                                                                          |
| <b>-MU</b>                      | Directs the assembler to perform a case-insensitive assembly, and to convert the names of external identifiers to uppercase when writing them to the object file. This is the default mode of operation for Microsoft MASM. |

<b>-MX</b>	Directs the assembler to perform a case-insensitive assembly, yet preserve the case of external identifiers when writing them to the object file.
<b>-N</b>	Prevents the symbol table summary from being printed in the listing file.
<b>-P</b>	Causes MASM to check for impure code operations when assembling for a privileged mode processor. This feature is not supported by ALP, thus MASM2ALP ignores this option.
<b>-S</b>	Forces MASM to write segments to the object file in source code order, nullifying the effects of any previously encountered <b>-A</b> option. MASM2ALP ignores this option.
<b>-X</b>	Forces false conditionals to be included in the listing file output.
<b>-Z</b>	Causes MASM to print the source line of any statement causing an assembly error to the standard output device. MASM2ALP ignores this option.
<b>-ZD</b>	Causes line number debugging information to be included in the object file output.
<b>-ZI</b>	Causes both line number and symbolic debugging information to be included in the object file output.

-----

## Language Elements

-----

## Description

The following sections describe the elements you use to build an ALP program source file.

### Character Set

All elements in an assembler language source file are built from collections of characters contained in the **character set**, which are defined as:

- The uppercase and lowercase letters of the English alphabet
- The decimal digits 0 through 9
- The following graphic characters:

```

~ ! " # $ % ^ & ' ( ) |
* + , - . / : ; = < > ?
[ \ ] _ { } @

```

- The space and horizontal tab characters
- The end of line character(s)

### White Space

White space is a character or contiguous stream of characters that is ignored or removed from the input stream by the ALP preprocessor.

**White space characters** are any contiguous sequence of one or more space or tab characters not enclosed in single or double quotes. White space characters are significant only in that they serve to separate language tokens from one another; they are removed from the input stream by the scanner.

-----

## Syntax

*Token :*

*Reserved-Word*

*Identifier*

*Literal*

*Punctuator*

---

## Reserved Words

---

## Description

This section describes all of the assembler reserved words.

---

## Syntax

*Reserved-Word :*

*Preprocessor-Directive*

*Assembler-Directive*

*Processor-Mnemonic*

*Processor-Register*

*Scalar-TypeName*

*Distance-TypeName*

*Language-Name*

*Anonymous-Label-Alias*

*Location-Counter-Alias*

*Indeterminate-Value-Alias*

*Directive-Keyword*

*Operator-Keyword*

---

## Preprocessor Directives

---

## Description

***Preprocessor Directives*** are symbolic names that describe the various assembly-time text processing instructions interpreted by the preprocessor phase of the assembler.

---

## Syntax

*Preprocessor-Directive* : one of

CATSTR	COMMENT	ELSE	ELSEIF
ELSEIF1	ELSEIF2	ELSEIFB	ELSEIFDEF
ELSEIFDIF	ELSEIFDIFI	ELSEIFE	ELSEIFIDN
ELSEIFIDNI	ELSEIFNB	ELSEIFNDEF	ENDIF
ENDM	EQU	EXITM	FOR
FORC	IF	IF1	IF2
IFB	IFDEF	IFDIF	IFDIFI
IFE	IFDN	IFIDNI	IFNB
IFNDEF	INCLUDE	INSTR	IRP
IRPC	LOCAL	MACRO	PURGE
REPEAT	REPT	SIZESTR	SUBSTR

# Assembler Directives

## Description

*Assembler Directives* are symbolic names that describe the various assembly-time instructions interpreted by the assembler itself.

## Syntax

*Assembler-Directive* : one of

.186	.286	.286C	.286P
.287	.386	.386C	.386P
.387	.486	.486C	.486P
.586	.586P	.686	.686P
.8086	.8087	ALIGN	.ALPHA
ASSUME	%BIN	.CODE	COMM
.CONST	.CREF	.DATA	.DATA?
DB	DD	DF	DOSSEG
.DOSSEG	DQ	DT	DW
ECHO	END	ENDP	ENDS
EQU	.ERR	.ERR1	.ERR2
.ERRB	.ERRDEF	.ERRDIF	.ERRDIFI
.ERRE	.ERRIDN	.ERRIDNI	.ERRNB
.ERRNDEF	.ERRNZ	EVEN	EXTERN
EXTERNDEF	EXTRN	.FARDATA	.FARDATA?
GROUP	INCLUDELIB	LABEL	.LALL
.LFCOND	.LIST	.LISTALL	.LISTIF
.LISTMACRO	.LISTMACROALL	LOCAL	.MMX
.MODEL	NAME	.NOCREF	.NOLIST
.NOLISTIF	.NOLISTMACRO	.NOMMX	.NOSIMD
OPTION	ORG	%OUT	PAGE
PROC	PUBLIC	.RADIX	RECORD
.SALL	SEGMENT	.SEQ	.SFCOND
.SIMD	.STACK	STRUC	STRUCT
SUBTITLE	SUBTTL	.TFCOND	TITLE
TYPEDEF	UNION	.XALL	.XCREF
.XLIST	.XMM		

---

# Processor Mnemonics

---

## Description

*Processor Mnemonics* are symbolic names given to the various instructions in the processor instruction set.

---

## Syntax

*Processor-Mnemonic* : one of

AAA	AAD	AAM	AAS
ADC	ADD	ADDPS	ADDSS
ANDNPS	ANDPS	ARPL	BOUND
BSF	BSR	BSWAP	BT
BTC	BTR	BTS	CALL
CBW	CDQ	CLC	CLD
CLI	CLTS	CMC	CMOVA
CMOVAE	CMOVB	CMOVBE	CMOVC
CMOVE	CMOVG	CMOVGE	CMOVL
CMOVLE	CMOVNA	CMOVNAE	CMOVNB
CMOVNBE	CMOVNC	CMOVNE	CMOVNG
CMOVNGE	CMOVNL	CMOVNLE	CMOVNO
CMOVNP	CMOVNS	CMOVNZ	CMOVO
CMOVP	CMOVPE	CMOVPO	CMOVS
CMOVZ	CMP	CMPPS	CMPEQPS
CMPLTPS	CMPLEPS	CMPUNORDPS	CMPNEQPS
CMPNLTPS	CMPNLEPS	CMPORDPS	CMPS
CMPSB	CMPSD	CMPSW	CMPSL
CMPEQSS	CMPLTSS	CMPLESS	CMPUNORDSS
CMPNEQSS	CMPNLTSS	CMPNLESS	CMPORDSS
CMPXCHG	CMPXCHG8B	COMISS	CPUID
CVTPI2PS	CVTSS2PI	CVTSI2SS	CVTSS2SI
CVTTPS2PI	CVTTSS2SI	CWD	CWDE
DAA	DAS	DEC	DIV
DIVPS	DIVSS	EMMS	ENTER
ESC	F2XM1	FABS	FADD
FADDP	FBLD	FBSTP	FCHS
FCLEX	FCMOVB	FCMOVBE	FCMOVE
FCMOVNB	FCMOVNBE	FCMOVNE	FCMOVNU
FCMOVU	FCOM	FCOMI	FCOMIP
FCOMP	FCOMPP	FCOS	FDECSTP
FDISI	FDIV	FDIVP	FDIVR
FDIVRP	FENI	FFREE	FIADD
FICOM	FICOMP	FIDIV	FIDIVR
FILD	FIMUL	FINCSTP	FINIT
FIST	FISTP	FISUB	FISUBR
FLD	FLDL	FLDCW	FLDENV
FLDENVD	FLDENVW	FLDL2E	FLDL2T
FLDLG2	FLDLN2	FLDPI	FLDZ
FMUL	FMULP	FNCLEX	FNDISI
FNENI	FNINIT	FNOP	FNRSTOR
FNSAVE	FNSAVED	FNSAVEW	FNSTCW
FNSTENV	FNSTENVW	FNSTSW	FNSTSW
FPTAN	FPREM	FPREM1	FPTAN
FRNDINT	FRSTOR	FRSTORD	FRSTORW
FSAVE	FSAVED	FSAVEW	FSCALE

FSETPM	FSIN	FSINCOS	FSQRT
FST	FSTCW	FSTENV	FSTENVDD
FSTENVW	FSTP	FSTSW	FSUB
FSUBP	FSUBR	FSUBRP	FTST
FUCOM	FUCOMI	FUCOMIP	FUCOMP
FUCOMPP	FWAIT	FXAM	FXCH
FXRSTOR	FXSAVE	FXTRACT	FYL2X
FYL2XP1	HLT	IDIV	IMUL
IN	INC	INS	INSB
INSD	INSW	INT	INTO
INVD	INVLPG	IRET	IRETD
IRETDF	IRETF	JA	JAE
JB	JBE	JC	JCXZ
JE	JECXZ	JG	JGE
JL	JLE	JMP	JNA
JNAE	JNB	JNBE	JNC
JNE	JNG	JNGE	JNL
JNLE	JNO	JNP	JNS
JNZ	JO	JP	JPE
JPO	JS	JZ	LAHF
LAR	LDMXCSR	LDS	LEA
LEAVE	LES	LFS	LGDT
LGS	LIDT	LLDT	LMSW
LOCK	LODS	LODSB	LODSD
LODSW	LOOP	LOOPD	LOOPE
LOOPED	LOOPEW	LOOPNE	LOOPNED
LOOPNEW	LOOPNZ	LOOPNZD	LOOPNZW
LOOPW	LOOPZ	LOOPZD	LOOPZW
LSL	LSS	LTR	MASKMOVQ
MAXPS	MAXSS	MINPS	MINSS
MOV	MOVAPS	MOVD	MOVHLPs
MOVHPS	MOVLHPS	MOVLPS	MOVMSKPS
MOVNTPSS	MOVNTQ	MOVQ	MOVSS
MOVSB	MOVSD	MOVSW	MOVSS
MOVSX	MOVUPS	MOVZX	MUL
MULPS	MULSS	NEG	NOP
ORPS	OUT	OUTS	OUTSB
OUTSD	OUTSW	PACKSSDW	PACKSSWB
PACKUSWB	PADDB	PADDD	PADDSB
PADDSW	PADDUSB	PADDUSW	PADDW
PAND	PANDN	PAVGB	PAVGW
PCMPEQB	PCMPEQD	PCMPEQW	PCMPGTB
PCMPGTD	PEXTRW	PINSRW	PCMPGTW
PMADDWD	PMAXSW	PMAXUB	PMINSW
PMINUB	PMOVMSKB	PMULHUW	PMULHW
PMULLW	POP	POPA	POPAD
POPD	POPF	POPFD	POPW
POR	PREFETCHT0	PREFETCHT1	PREFETCHT2
PREFETCHNTA	PSADBW	PSHUFW	PSLLD
PSLLQ	PSLLW	PSRAD	PSRAW
PSRLD	PSRLQ	PSRLW	PSUBB
PSUBD	PSUBSB	PSUBSW	PSUBUSB
PSUBUSW	PSUBW	PUNPCKHBW	PUNPCKHDQ
PUNPCKHWD	PUNPCKLBW	PUNPCKLDQ	PUNPCKLWD
PUSH	PUSHA	PUSHAD	PUSHD
PUSHF	PUSHFD	PUSHW	PXOR
RCL	RCPPS	RCPSS	RCR
RDMSR	RDPMC	RDTSC	REP
REPE	REPNE	REPNZ	REPZ
RET	RETF	RETN	ROL
ROR	RSM	RSQRTPS	RSQRTSS
SAHF	SAL	SAR	SBB
SCAS	SCASB	SCASD	SCASW
SETA	SETAE	SETB	SETBE
SETC	SETE	SETG	SETGE
SETL	SETLE	SETNA	SETNAE
SETNB	SETNBE	SETNC	SETNE
SETNG	SETNGE	SETNL	SETNLE
SETNO	SETNP	SETNS	SETNZ
SETO	SETP	SETPE	SETPO
SETS	SETZ	SFENCE	SGDT
SHLD	SHRD	SHUFFPS	SIDT
SLDT	SMSW	SQRTPS	SQRTSS
STC	STD	STI	STMXCSR
STOS	STOSB	STOSD	STOSW
STR	SUB	SUBPS	SUBSS
SYSEXIT	SYSEXIT	TEST	UCOMISS
UD2	UNPCKHPS	UNPCKLPS	VERR
VERW	WAIT	WBINVD	WRMSR
XADD	XCHG	XLAT	XLATB
XORPS			

---

# Processor Registers

---

## Description

**Processor Registers** are the symbolic names assigned to the various internal processor registers. They are normally used as operands to processor instructions.

---

## Syntax

*Processor-Register :*

*General-Purpose-Register*

*Segment-Register*

*Control-Register*

*Debug-Register*

*Test-Register*

*MMX-Register*

*Floating-Point-Register*

*General-Purpose-Register :*

*8-Bit-Register*

*16-Bit-Register*

*32-Bit-Register*

*8-Bit-Register :* one of

**AL AH BL BH CL CH DL DH**

*16-Bit-Register :* one of

**AX BX CX DX DI SI BP SP**

*32-Bit-Register :* one of

**EAX EBX ECX EDX EDI ESI EBP ESP**

*Segment-Register :* one of

**CS DS ES FS GS SS**

*Control-Register :* one of

**CR0 CR2 CR3 CR4**

*Debug-Register :* one of

**DR0 DR1 DR2 DR3 DR4 DR5 DR6 DR7**

*Test-Register :* one of

**TR3 TR4 TR5 TR6 TR7**

*MMX-Register :* one of

**MM0 MM1 MM2 MM3 MM4 MM5 MM6 MM7**

*Floating-Point-Register :*

**ST**

*SIMD-Register :* one of

**XMM0 XMM1 XMM2 XMM3 XMM4 XMM5 XMM6 XMM7**

---

# Scalar Type Names

---

## Description

***Scalar Type Names*** are the symbolic names given to the integral data types. These are the fundamental types of data upon which the processor can directly operate.

---

## Syntax

*Scalar-TypeName :*  
BYTE  
SBYTE  
WORD  
SWORD  
DWORD  
SDWORD  
REAL4  
FWORD  
QWORD  
REAL8  
TBYTE  
REAL10

---

# Distance Type Names

---

## Description

***Distance Type Names*** are the symbolic names given to the integral types of pointers directly supported by the processor. Their names reflect a fundamental property of the Intel processor architecture known as *distance*. The type of pointer is defined by the *distance* required to reach the information to which it points.

---

## Syntax

*Distance-TypeName :*  
NEAR  
NEAR16  
NEAR32

FAR  
FAR16  
FAR32

-----

## Language Names

-----

## Description

*Language Names* refer to the various high level programming languages (or more specifically, the calling conventions used by such languages) with which the assembler has the ability to interface.

-----

## Syntax

*Language-Name* :

C  
SYSCALL  
STDCALL  
PASCAL  
FORTRAN  
BASIC  
OPTLINK

-----

## Anonymous Label Aliases

-----

## Description

The *Anonymous Label Aliases* are reserved symbolic names that return a context-sensitive value when referenced in expressions.

The reserved name **@B** (backward reference) returns the internally generated name representing the nearest **@@:**code label appearing before the current location in the input stream.

The reserved name **@F** (forward reference) returns the internally generated name representing the nearest **@@:**code label appearing after the current location in the input stream.

-----

## Syntax

*Anonymous-Label-Alias :*

@B  
@F

-----

## Location Counter Alias

-----

## Description

The ***Location Counter Alias*** is a reserved name used in expressions to return the offset within the current segment or structure being assembled.

-----

## Syntax

*Location-Counter-Alias :*

\$

-----

## Indeterminate Value Alias

-----

## Description

The ***Indeterminate Value Alias*** is a reserved name used in expressions to represent an uninitialized value.

-----

## Syntax

*Indeterminate-Value-Alias :*

?

-----

## Directive Keywords

---

## Description

***Directive Keywords*** are symbolic names recognized and used in the body of various assembler directives.

---

## Syntax

*Directive-Keyword* :

ABS	AT	BASIC	C
CASEMAP	CODE	COMMON	DOTNAME
EMULATOR	EPILOGUE	ERROR	EXPORT
EXPR16	EXPR32	FARSTACK	FLAT
FORTTRAN	HUGE	LANGUAGE	LARGE
LJMP	MEDIUM	NEARSTACK	NODOTNAME
NOEMULATOR	NOKEYWORD	NOLANGUAGE	NOLJMP
NONE	NOOLDMACROS	NOOLDSTRUCTS	NOREADONLY
NOSCOPED	NOSIGNEXTEND	NOTHING	NOTPUBLIC
OLDMACROS	OLDSTRUCTS	OPTLINK	OS_DOS
OS_OS2	PAGE	PARA	PASCAL
PRIVATE	PROC	PROLOGUE	PUBLIC
READONLY	SCOPED	SEGMENT	SIGNEXTEND
SMALL	STACK	STDCALL	SYSCALL
TINY	USE16	USE32	USES

---

## Operator Keywords

---

## Description

***Operator Keywords*** are symbolic names used in expressions to denote an operation to be performed on one or more operands.

---

## Syntax

*Operator-Keyword* :

AND	DUP	EQ	GE
GT	HIGH	HIGHWORD	LE
LENGTH	LENGTHOF	LOW	LOWWORD
LT	MASK	MOD	NE
NOT	OFFSET	OPATTR	OR

PTR	SEG	SHL	SHORT
SHR	SIZE	SIZEOF	THIS
.TYPE	TYPE	WIDTH	XOR

---

# Identifiers

---

## Description

This section describes the syntax for identifiers and the various types of information they can be made to represent.

---

## Syntax

*Identifier* :

*Normal-Identifier*

*Dot-Identifier*

*Normal-Identifier*

*NonDigit*

*Normal-Identifier Identifier-Character*

*Dot-Identifier*

*. Normal-Identifier*

*Identifier-Character*

*NonDigit*

*Digit*

*NonDigit*: one of

\_ \$ @ ?

a b c d e f g h i j k l m

n o p q r s t u v w x y z

A B C D E F G H I J K L M

N O P Q R S T U V W X Y Z

*Digit*: one of

0 1 2 3 4 5 6 7 8 9

---

# Identifier Types

---

## Description

This section describes the various types of identifiers that the assembler will create and manipulate.

---

## Definition

*Identifier-Type :*  
*EquateName*  
*FieldName*  
*GroupName*  
*LabelName*  
*MacroName*  
*SegmentName*  
*UserDefined-TypeName*

---

## Equate Name

### Definition

*EquateName :*  
*Numeric-EquateName*  
*Text-EquateName*

### Description

An *EquateName* is a symbolic identifier that is associated with an expression or a body of text. The assembler substitutes the value of the *EquateName* at the point of reference.

---

## Numeric Equate Name

An identifier becomes a *Numeric-EquateName* when it is defined in a **EQU** or **=** directive. Procedure **parameter** names and **local variable** names are also created as *Numeric-EquateName*s, but are visible only from within the procedure where they are defined. All other *Numeric-EquateName*s are globally-scoped identifiers visible across the entire module.

A *Numeric-EquateName* may only be referenced from within **expressions**, as its replacement value is itself an expression.

---

## Text Equate Name

A *Text-EquateName* is a globally-scoped identifier created during the processing of a [EQU](#) preprocessor directive. A *Text-EquateName* is associated with a body of text whose content may not span across line breaks. In certain contexts the assembler replaces the *Text-EquateName* with the text that it represents and recursively evaluates the result.

---

## Field Name

### Definition

*FieldName* :  
[Record-FieldName](#)  
[Structure-FieldName](#)  
[Union-FieldName](#)

### Description

An identifier becomes a *FieldName* when it is defined within a [RECORD](#), [STRUCT](#), or [UNION](#) directive.

---

## Record Field Name

A *Record-FieldName* is a globally-scoped identifier created during the processing of a [RECORD](#) directive. It is a special variation of a [Numeric-EquateName](#) and can be used in the same contexts.

---

## Structure Field Name

An identifier becomes a *Structure-FieldName* when it is defined in a [STRUCT](#) directive. If the assembler is operating in [M510](#) mode, or if the [OPTION OLDSTRUCTS](#) directive has been specified, then a *Structure-FieldName* is a globally-scoped identifier treated as a special variation of a [Numeric-EquateName](#) and can be used in the same contexts. Otherwise, a *Structure-FieldName* is private to the defining structure and is only accessible in expressions through use of the [Structure/Union Field Selection \(. Operator\)](#).

---

## Union Field Name

An identifier becomes a *Union-FieldName* when it is defined in a [UNION](#) directive. A *Union-FieldName* is private to the defining union and is only accessible in expressions through use of the [Structure/Union Field Selection \(. Operator\)](#).

---

## Group Name

A *GroupName* is a globally-scoped identifier created during the processing of a [GROUP](#) directive. It is referenced from within expressions.

---

## Label Name

### Definition

*Label/Name* :  
*Code-Label/Name*  
*Data-Label/Name*

## Description

A *Label/Name* is globally-scoped identifier that is associated with a program address at application run-time. It has an explicit or inherited *Type-Declaration*, and an optional *Language-Attribute*. These attributes are described in the following sections.

### Type Declaration

The type declaration associated with a label name depends on how the label was defined. See the *Code-Label/Name* and *Data-Label/Name* sections for descriptions on how this attribute is assigned.

### Language Attribute

A *Label/Name* can have an assigned *Language-Attribute*, set either implicitly through the use of a *Language-Name* keyword in the body of a *.MODEL* or *OPTION* directive, or explicitly through the use of an overriding *Language-Name* keyword in the body of a *EXTERN/EXTRN*, *EXTERNDEF*, *PROC*, or *PUBLIC* directive. The *Language-Attribute* determines the exact spelling of the *Label/Name* identifier when it is written to the object file. According to the *Language-Attribute*, identifier spellings are modified from their appearance in the assembly language source module as follow:

LANGUAGE ATTRIBUTE IDENTIFIER SPELLING	
OPTLINK, SYSCALL	No modifications are made to the identifier when written to the object file.
C, STDCALL	A leading underscore character is appended to the front of the name.
BASIC, FORTRAN, PASCAL	All characters in the identifier are converted to uppercase.

## Code Label Name

### Definition

*Code-Label/Name* :  
*Target-Label/Name*  
*Procedure-Label/Name*

### Description

A *Code-Label/Name* is an identifier that is associated with an executable code address at application run-time. There are two types of *Code-Label/Name* s : *Target-Label/Name* s and *Procedure-Label/Name* s .

## Target Label Name

An identifier becomes a *Target-Label/Name* when it is defined with a *:*, *::*, or *LABEL* directive.

If a *Target-Label/Name* created with a single colon (*:*) is defined within the body of a *procedure*, then the name is visible only from within that procedure unless operating in *M510* mode (and no *.MODEL* directive with a *Language-Name* has been specified), or unless the *OPTION NOSCOPED* directive has been specified.

A *Target-LabelName* defined outside the body of a procedure is visible to the entire module, and may also be given **PUBLIC** visibility.

---

## Procedure Label Name

An identifier becomes a *Procedure-LabelName* when it is defined in a **PROC** directive.

---

## Data Label Name

A *Data-LabelName* is an identifier that is the address of a program variable at application run-time. An identifier becomes a *Data-LabelName* when it is named in a data allocation statement, or when a scalar, aggregate, or vector type is associated with the identifier named in a **LABEL**, **EXTERN/EXTRN**, **EXTERNDEF**, or **COMM** directive.

---

## Macro Name

A *MacroName* is a globally-scoped identifier created during the processing of a **MACRO** directive. It is associated with a multi-line body of text. A *MacroName* may only be used in contexts where a normal assembler directive is expected.

---

## Macro Parameter Name

An identifier becomes a *Macro-ParameterName* when it is named as a parameter to a macro in a **MACRO** directive. It is associated with a body of text whose content may not span across line breaks. It is only recognized and acted upon from within the body of a macro expansion.

---

## Segment Name

A *SegmentName* is a globally-scoped identifier created during the processing of a **SEGMENT** directive. It may be referenced from within expressions or in the body of a **GROUP** directive.

---

## User-Defined Type Name

### Definition

*UserDefined-TypeName* :  
*Record-TypeName*  
*Structure-TypeName*  
*Typedef-TypeName*  
*Union-TypeName*

### Description

An identifier becomes a *UserDefined-TypeName* when it is defined within a **RECORD**, **STRUCT**, **TYPDEF**, or **UNION** directive.

---

# Record Type Name

A *Record-TypeName* is a globally-scoped identifier created during the processing of a **RECORD** directive. It is recognized from within *Expression s*, *Type-Declaration s*, or as a pseudo-directive in a [data allocation statement](#).

---

# Structure Type Name

A *Structure-TypeName* is a globally-scoped identifier created during the processing of a **STRUCT** directive. It is recognized from within *Expression s*, *Type-Declaration s*, or as a pseudo-directive in a [data allocation statement](#).

---

# Typedef Type Name

A *Typedef-TypeName* is a globally-scoped identifier created during the processing of a **TYPDEF** directive. It is recognized from within *Expression s*, *Type-Declaration s*, or as a pseudo-directive in a [data allocation statement](#).

---

# Union Type Name

A *Union-TypeName* is a globally-scoped identifier created during the processing of a **UNION** directive. It is recognized from within *Expression s*, *Type-Declaration s*, or as a pseudo-directive in a [data allocation statement](#).

---

# Predefined Identifiers

The following sections describe the predefined identifiers created by the assembler. When a case-sensitive assembly is being performed, the predefined identifiers must be spelled exactly as they appear in the following descriptions with respect to uppercase and lowercase characters.

---

# Segment Information

The following sections describe the predefined identifiers created by the assembler in support of segment manipulation.

---

## @code

The **@code** identifier is a *Text-EquateName* created by the assembler when a **.MODEL** directive is encountered, at which time the assembler performs an automatic **ASSUME CS: @code** operation. The **@code** symbol is not defined if a **.MODEL** directive has not been issued.

Under MASM 5.10 emulation, the **@code** symbol is set to the name of the implicitly-defined default code segment (the segment opened when a **.CODE** directive is used) and its value is never changed. In other modes, the **@code** symbol is updated to reflect whatever segment is opened by using **.CODE**, whether defined implicitly or as an explicit parameter to the **.CODE** directive.

The value assigned to the **@code** symbol when the default code segment is opened is determined by the memory model as follows:

Memory Model	Value for @code
TINY	DGROUP
SMALL	__TEXT
MEDIUM	module__TEXT
COMPACT	__TEXT
LARGE	module__TEXT
HUGE	module__TEXT
FLAT	CODE32

The *module* entry is replaced with base file name of the top-level module being assembled.

## @CodeSize

The **@CodeSize** identifier is a *Numeric-EquateName* created by the assembler when a **.MODEL** directive is encountered. **@CodeSize** indicates whether code segments created by the **.CODE** directive are named such that the linker will combine them into a single (NEAR) segment or into multiple (FAR) segments. The **@CodeSize** symbol is set to 0 (NEAR) for the **TINY**, **SMALL**, **COMPACT**, and **FLAT** memory models, and to 1 (FAR) for the **MEDIUM**, **LARGE**, and **HUGE** memory models. The **@CodeSize** symbol is not defined if a **.MODEL** directive has not been issued.

## @CurSeg

The **@CurSeg** identifier is a *Text-EquateName* defined by the assembler to hold the name of the currently opened segment. If no segment is currently open, **@CurSeg** will expand into an empty string.

## @data

The **@data** identifier is a *Text-EquateName* created by the assembler when a **.MODEL** directive is encountered. It expands to the group name shared by all of the near data segments. If a **.MODEL FLAT** has been issued, the **@data** identifier expands to FLAT. For all other memory models, it expands to DGROUP.

## @DataSize

The **@DataSize** identifier is a *Numeric-EquateName* created by the assembler when a **.MODEL** directive is encountered, and represents the default data distance. Depending on the currently selected memory model, the **@DataSize** identifier is set to the following values:

TINY	0
SMALL	0
MEDIUM	0
COMPACT	1
LARGE	1
HUGE	2
FLAT	0

## @Model

The **@Model** identifier is a *Numeric-EquateName* created by the assembler when a **.MODEL** directive is encountered, and is set to a unique value for each memory model. The values are as follows:

TINY	1
SMALL	2
COMPACT	3
MEDIUM	4
LARGE	5
HUGE	6
FLAT	7

-----

## @WordSize

The **@WordSize** identifier is a *Numeric-EquateName* that reflects the address size attribute of the current segment. It is set to 2 for a **USE16** segment, and 4 for a **USE32** segment. If no segment is currently open, it reflects the default address size as determined by the currently selected processor.

-----

## Version Information

These identifiers offer methods of testing the various operating modes of the assembler to determine what features are activated or disabled, or how the assembler will behave under various conditions.

-----

## @Alp

The **@Alp** identifier is a *Text-EquateName* that can be tested to determine if ALP is assembling the source file (versus some other assembler). It is always set to the string **100**.

-----

## @AlpMajor

The **@AlpMajor** identifier is a *Text-EquateName* that reflects the *major* portion of the three-part assembler version number. It is padded on the right with zeros to allow major version number comparisons independant of the minor version and revisions numbers. See [@AlpVersion](#) for more information.

This identifier is only defined in [ALP](#) mode.

-----

## @AlpMinor

The **@AlpMinor** identifier is a *Text-EquateName* that reflects the *minor* portion of the three-part assembler version number. It is padded on the right with zeros to allow minor version number comparisons independant of the major version and revisions numbers. See [@AlpVersion](#) for more information.

This identifier is only defined in [ALP](#) mode.

-----

## @AlpRevision

The **@AlpRevision** identifier is a *Text-EquateName* that reflects the *revision* portion of the three-part assembler version number. It allows

revision number comparisons independent of the major and minor version numbers. See [@AlpVersion](#) for more information.

This identifier is only defined in [ALP](#) mode.

---

## @AlpVersion

The **@AlpVersion** identifier is a *Text-EquateName* that reflects the full three-part assembler version number. This is an encoding of the version number printed in the program banner when the assembler is invoked. This number and its requisite parts may be tested to determine the presence or absence of features provided by the assembler.

The assembler version number consists of three parts:

1. The major version number (one digit)
2. The minor version number (two digits)
3. The revision number (three digits)

In the assembler banner, the numbers are separated by the period (.) character; the period is removed from the text defined by the predefined identifiers.

For example, if the major version number is **1**, the minor version number is **2**, and the revision number is **3**, then the full version number is printed in the assembler banner as **1.02.003**, and the various predefined version identifiers would be set as follows:

```
@AlpVersion    102003
@AlpMajor      100000
@AlpMinor      2000
@AlpRevision    003
```

This identifier is only defined in [ALP](#) mode.

---

## @Cpu

The **@Cpu** identifier is a *Numeric-EquateName* that reflects the currently selected processor for which ALP is assembling instructions. This value is affected by issuing a *Processor-Control-Directive*, and is a bit map that indicates the currently active processor instruction set(s).

B A 9 8 7 6 5 4 3 2 1 0 BIT SET IF ASSEMBLING FOR:

	1	8086/8088
	1	80186
	1	80286
	1	80386
	1	80486
	1	80586 (Pentium)
	1	80686 (Pentium Pro)
	1	Privileged mode
	1	8087
	1	MMX Extensions
	1	80287
	1	80387

## @Version

The **@Version** identifier is a *Text-EquateName* that reflects the MASM-compatible version number. The current emulation mode of the assembler affects the value of this symbol as follows:

M510	510
M600	600
M611	611
M612	612
M613	613
M614	614
ALP	4294967295 (the highest possible value for an unsigned 32-bit integer)

-----

## Date and Time Information

These identifiers allow the programmer to query the system date or time during the assembly. Each time they are referenced, a new system request for the current date and time is made and the values held in the identifiers are refreshed.

-----

### @Date

The **@Date** identifier is a *Text-EquateName* that is set to the current system date. If the current operating mode is M600, the date is returned in the MM/DD/YY format. In native ALP mode, the date is returned in the MM/DD/YYYY format.

The **@Date** identifier is not available in M510 mode.

-----

### @Time

The **@Time** identifier is a *Text-EquateName* that is set to the current system time in 24-hour HH:MM:SS format.

The **@Time** identifier is not available in M510 mode.

-----

## File Information

These identifiers return information about the file(s) being assembled.

-----

### @FileName

The **@FileName** identifier is a *Text-EquateName* that is set to the base name of the main file being assembled (as it appears on the command line).

-----

### @Line

The **@Line** identifier is a *Numeric-EquateName* that is set to the current source line number in the file currently being assembled.

The **@Line** identifier is not available in **M510** mode.

---

## Literals

---

## Description

**Literals** are the notational method whereby numeric values or strings of character data are represented in the source stream. Literals are also commonly referred to as **constants** (especially in the context of high level languages) because they typically represent objects whose values do not change throughout the life of the assembly or compilation. However, literals should not be confused with run-time "constants" ("read-only" data items allocated by the programmer); they are assembly-time tokens used by the assembler to represent numeric values or character strings.

---

## Syntax

*Literal :*  
*Floating-Point-Literal*  
*Integer-Literal*  
*String-Literal*

---

## Integer Literals

---

## Description

An **integer literal** represents a fixed-point numeric value. An integer literal must begin with one of the numeric digits 0 - 9, and may be optionally terminated with a suffix character called a **radix specifier**. The radix specifier tells the assembler whether the literal is to be interpreted as a base 2 (binary), 8 (octal), 10 (decimal), or 16 (hexadecimal) number. If the literal is not suffixed with a radix specifier, the assembler uses the value of the current radix to determine the base of the number. The default radix is 10 (decimal), but the **.RADIX** directive can be used to specify an alternate radix.

---

## Syntax

*Integer-Literal :*  
*Binary-Integer-Literal*

---

## Binary Integer Literals

---

## Related Information

[Description](#)

[Examples](#)

---

## Syntax

```
Binary-Integer-Literal :  
    Unqualified-Binary-Integer-Literal  
    Qualified-Binary-Integer-Literal  
  
Unqualified-Binary-Integer-Literal:  
    Binary-Digit  
    Binary-Integer-Literal Binary-Digit  
  
Qualified-Binary-Integer-Literal:  
    Unqualified-Binary-Integer-Literal Binary-Radix  
  
Binary-Digit:  
    0  
    1  
  
Binary-Radix:  
    b  
    B  
    y  
    Y
```

---

## Description

A base-2 number containing either of the digits *0* and *1*.

---

## Examples

The following are examples of unqualified binary integer literals:

10101

0  
000001  
1111000010101010

The following are examples of qualified binary integer literals:

00001111b  
1111Y  
00y  
1111000010101010B

---

## Octal Integer Literals

---

## Related Information

[Description](#)

[Examples](#)

---

## Syntax

*Octal-Integer-Literal* :  
    *Unqualified-Octal-Integer-Literal*  
    *Qualified-Octal-Integer-Literal*

*Unqualified-Octal-Integer-Literal* :  
    *Octal-Digit*  
    *Octal-Integer-Literal Octal-Digit*

*Qualified-Octal-Integer-Literal* :  
    *Unqualified-Octal-Integer-Literal Octal-Radix*

*Octal-Digit*: one of:  
    **0 1 2 3 4 5 6 7**

*Octal-Radix*:  
    o  
    O  
    q  
    Q

---

## Description

A base-8 number containing any of the digits *0* through *7*.

---

## Examples

The following are examples of unqualified octal integer literals:

01234567  
27  
765

The following are examples of qualified octal integer literals:

27q  
013o  
567O  
01234567Q

---

## Decimal Integer Literals

---

## Related Information

[Description](#)

[Examples](#)

---

## Syntax

*Decimal-Integer-Literal :*

*Unqualified-Decimal-Integer-Literal*  
*Qualified-Decimal-Integer-Literal*

*Unqualified-Decimal-Integer-Literal:*

*Decimal-Digit*  
*Decimal-Integer-Literal Decimal-Digit*

*Qualified-Decimal-Integer-Literal:*

*Unqualified-Decimal-Integer-Literal Decimal-Radix*

*Decimal-Digit:* one of:

**0 1 2 3 4 5 6 7 8 9**

*Decimal-Radix:*

**d  
D  
t  
T**

---

## Description

A base-10 number containing any of the digits *0* through *9*.

---

# Examples

The following are examples of unqualified decimal integer literals:

0123456789  
19  
090

The following are examples of qualified decimal integer literals:

01d  
89t  
4567D  
0123456789T

## Hexadecimal Integer Literals

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

## Syntax

*Hexadecimal-Integer-Literal :*

*Unqualified-Hexadecimal-Integer-Literal*  
*Qualified-Hexadecimal-Integer-Literal*

*Unqualified-Hexadecimal-Integer-Literal:*

*Decimal-Digit*  
*Hexadecimal-Integer-Literal Decimal-Digit*  
*Hexadecimal-Integer-Literal Hexadecimal-Digit*

*Qualified-Hexadecimal-Integer-Literal:*

*Unqualified-Hexadecimal-Integer-Literal Hexadecimal-Radix*

*Decimal-Digit:* one of:

**0 1 2 3 4 5 6 7 8 9**

*Hexadecimal-Digit:* one of:

**a b c d e f**  
**A B C D E F**

*Hexadecimal-Radix:*

**h**  
**H**

# Description

A base-16 number using any combination of the digits **0** through **9** and the lowercase letters **a** through **f** or the uppercase letters **A** through **F**. The lowercase and uppercase representations of any given hexadecimal letter are equivalent.

---

# Constraints

A hexadecimal integer literal may not begin with any of the alphabetic hexadecimal characters or it will be interpreted as an identifier; such numbers must be prefixed with the *0* digit.

---

# Examples

The following are examples of unqualified hexadecimal integer literals:

01BD  
9A  
0AB

The following are examples of qualified hexadecimal integer literals:

1234ABCDh  
01DH  
0bh  
1111FFFFH

---

# Floating-Point Literals

---

# Description

A *floating-point literal* is a notation for representing real numbers. The assembler provides both decimal and hexadecimal floating-point notations for representing real numbers.

---

# Syntax

*Floating-Point-Literal* :  
*Decimal-Floating-Point-Literal*  
*Hexadecimal-Floating-Point-Literal*

---

# Decimal Floating-Point Literals

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

```
Decimal-Floating-Point-Literal :  
    Significand-Part  
    Significand-Part Exponent-Part  
  
Significand-Part:  
    Digit-Sequence . Digit-Sequence  
    Digit-Sequence .  
  
Exponent-Part:  
    E-Character Digit-Sequence  
    E-Character Sign Digit-Sequence  
  
E-Character:  
    e  
    E  
  
Sign:  
    -  
    +  
  
Digit-Sequence:  
    Digit  
    Digit-Sequence Digit  
  
Digit: one of:  
    0 1 2 3 4 5 6 7 8 9
```

---

## Description

A decimal floating-point literal has a *significand part* that may be followed by an *exponent part*. The significand part consists of a digit sequence representing the whole-number part, followed by a period (.), followed by a digit sequence representing the fraction part. The exponent part consists of an introductory character (**e** or **E**), followed by an optional sign character (**+** or **-**), followed by a digit sequence representing the exponent.

---

## Constraints

The introductory *Digit-Sequence* in the *Significand-Part* must be specified (the literal cannot begin with a ".").

---

## Examples

```
25.23
2.523E1
2523.0E-2
```

---

## Hexadecimal Floating-Point Literals

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

```
Hexadecimal-Floating-Point-Literal :
    Hexadecimal-Literal Float-Radix

Hexadecimal-Literal:
    Decimal-Digit
    Hexadecimal-Literal Decimal-Digit
    Hexadecimal-Literal Hexadecimal-Digit

Decimal-Digit: one of:
    0 1 2 3 4 5 6 7 8 9

Hexadecimal-Digit: one of:
    a b c d e f
    A B C D E F

Float-Radix:
    r
    R
```

---

## Description

A hexadecimal floating-point literal provides a means of initializing floating point values using a notation more closely tied to the internal machine representation than that of the [Decimal-Floating-Point-Literal](#). Such literals are coded in a fashion similar to that of a normal [Hexadecimal-Integer-Literal](#), but a different radix suffix is used to inform the assembler that the value is to be used in the allocation of real numbers rather than integers.

-----

## Constraints

A hexadecimal floating-point literal may not begin with any of the alphabetic hexadecimal characters or it will be interpreted as an identifier; such numbers must be prefixed with the *0* digit.

The literal must specify the correct number of hexadecimal digits according to the size of the real-number data-type to which it will be assigned. For **REAL4**, **REAL8**, and **REAL10** variables, the respective number of digits in the literal must be 8, 16, and 20. For literals encoded with a leading zero, the respective number of digits must be 9, 17, and 21.

-----

## Examples

3F800000r

-----

## String Literals

-----

## Related Information

[Description](#)

[Examples](#)

-----

## Syntax

*String-Literal :*

*D-String*  
*S-String*

*D-String :*

*D-Quote D-Quote*  
*D-Quote D-Char-Sequence D-Quote*

*S-String :*

*S-Quote S-Quote*  
*S-Quote S-Char-Sequence S-Quote*

*D-Char-Sequence* :  
any printable character except *D-Quote*  
*D-Quote D-Quote*

*S-Char-Sequence* :  
any printable character except *S-Quote*  
*S-Quote S-Quote*

*D-Quote* :  
"

*S-Quote* :  
,

-----

## Description

A *string literal* contains a sequence of zero or more characters enclosed in quotation mark symbols. Either a single (') or double (") quotation mark symbol may be used as the *quote character* that opens and closes the string literal. If a single quotation mark symbol is used as the quote character, then double quotation mark symbols may appear as data characters within the string literal, and vice versa. If the quote character must also appear as a character within the string literal, use two adjacent quote characters; this will allow a single occurrence of the quote character to be inserted into the string literal.

A quote character must be used to terminate the string literal before the end of the line is reached, otherwise an error message is issued and the literal is terminated by the end of line character. A string literal may span multiple lines only if a backslash (\) appears as the last non-whitespace character on the line, in which case the backslash, all surrounding whitespace characters, and the end of line character are deleted and the literal is continued with the first character on the next line.

-----

## Examples

```
'Hello, world'  
"That's the way it is"  
'Unless it''s not'  
"SuperStringCon \  
catenated"
```

-----

## Punctuators

-----

## Description

Punctuators are used as operators and separator characters.

-----

## Syntax

*Punctuator* : one of  
[ ] ( ) { } \* , : = ; %

---

## Declarations

A *Type Declaration* is a language construct that specifies the characteristics of code and data objects used in a program.

---

## Type Declarations

## Description

A *Type-Declaration* is a common construct used in various assembler directives to establish type attribute information for a program object. A *Type-Declaration* is needed to determine the data type of a variable or labeled address. The [TYPEDEF](#) directive offers a method of assigning a name to a *Type-Declaration*.

---

## Syntax

*Type-Declaration* :  
    *TypeName*  
    *TypeName* *Array-Spec*  
    *Pointer-Spec*  
    *Pointer-Spec* *TypeName*  
    *Pointer-Spec* *TypeName* *Array-Spec*

*Pointer-Spec* :  
    **PTR**  
    *Distance-TypeName* **PTR**  
    *Pointer-Spec* *Array-Spec*

*Array-Spec* :  
    [ *Expression* ]  
    *Array-Spec* [ *Expression* ]

*TypeName* :  
    *Distance-TypeName*  
    *Scalar-TypeName*  
    *UserDefined-TypeName*

---

## Examples

The **TYPEDEF** directive is used to illustrate the type declaration syntax:

```

CHAR      typedef byte          ; Alias of intrinsic TypeName
PBYTE     typedef ptr byte      ; Pointer to intrinsic TypeName
PCHAR     typedef ptr CHAR      ; Pointer to TypeDef-TypeName
PPCHAR    typedef ptr PCHAR     ; Pointer to a pointer to a CHAR
PPBYTE    typedef ptr ptr byte  ; Similar to PPCHAR
PVOID     typedef ptr          ; Pointer to nothing (pointer to code)
PCODE     typedef ptr PROC      ; Similar to PVOID
PFCODE    typedef far ptr far   ; Far pointer to far code address

; vector declarations

ACHAR     typedef CHAR[16]      ; Array of 16 characters
AAWORD    typedef word[2][2]    ; multi-dimensional array
APBYTE    typedef ptr[8] byte   ; Array of 8 pointers to byte
APACHAR   typedef ptr[4] ACHAR  ; Array of 4 ptrs to arrays of 16 chars

SIZES_T   struct                ; define an intrinsic structure type
    little byte    ?
    Medium word    ?
    BIG dword     ?
SIZES_T   ends

SIZES     typedef SIZES_T       ; alias for intrinsic structure type
PSIZES    typedef ptr SIZES_T   ; and a type to point to it

PFORWARD  typedef ptr FORWARD  ; Pointers to forward-referenced types
FORWARD    struct                ; are assumed to be pointers to structs
    blah word      ?
FORWARD    ends

```

## Expressions

An expression is a sequence of *operators* and *operands* that are evaluated to derive a numeric result, an effective address, or a register operand.

Expressions are specified using standard infix notation, which is recursive in nature, ie., expressions may be nested within other expressions. The evaluation of an expression occurs in a left to right manner, and is influenced by the rules of operator *precedence* and *associativity*. The order in which expressions are evaluated can be controlled by grouping operands and operators together using parentheses ().

## Expression Syntax

## Description

This section describes the complete expression syntax.

## Syntax

*Expression :*  
*Duplicative-Expression*  
*Duplicative-Expression :*

*Attribute-Expression*  
*Attribute-Expression* DUP ( *Initializer-List* )

*Attribute-Expression* :  
    *OR-Expression*  
    SHORT *Additive-Expression*  
    .TYPE *OR-Expression*  
    OPATTR *OR-Expression*

*OR-Expression* :  
    *AND-Expression*  
    *OR-Expression* OR *AND-Expression*  
    *OR-Expression* XOR *AND-Expression*

*AND-Expression* :  
    *NOT-Expression*  
    *AND-Expression* AND *NOT-Expression*

*NOT-Expression* :  
    *Relational-Expression*  
    NOT *Relational-Expression*

*Relational-Expression* :  
    *Additive-Expression*  
    *Relational-Expression* EQ *Additive-Expression*  
    *Relational-Expression* NE *Additive-Expression*  
    *Relational-Expression* GT *Additive-Expression*  
    *Relational-Expression* GE *Additive-Expression*  
    *Relational-Expression* LT *Additive-Expression*  
    *Relational-Expression* LE *Additive-Expression*

*Additive-Expression* :  
    *Multiplicative-Expression*  
    *Additive-Expression* + *Multiplicative-Expression*  
    *Additive-Expression* - *Multiplicative-Expression*

*Multiplicative-Expression* :  
    *Narrowed-Expression*  
    *Multiplicative-Expression* \* *Narrowed-Expression*  
    *Multiplicative-Expression* | *Narrowed-Expression*  
    *Multiplicative-Expression* MOD *Narrowed-Expression*  
    *Multiplicative-Expression* SHL *Narrowed-Expression*  
    *Multiplicative-Expression* SHR *Narrowed-Expression*

*Narrowed-Expression* :  
    *Cast-Expression*  
    HIGH *Cast-Expression*  
    HIGHWORD *Cast-Expression*  
    LOW *Cast-Expression*  
    LOWWORD *Cast-Expression*

*Cast-Expression* :  
    *Element-Selection-Expression*  
    OFFSET *Cast-Expression*  
    SEG *Cast-Expression*  
    THIS *Element-Selection-Expression*  
    TYPE *Element-Selection-Expression*  
    *Cast-Expression* PTR *Cast-Expression*  
    *Cast-Expression* : *Cast-Expression*

*Element-Selection-Expression* :  
    *Sign-Expression*  
    *Element-Selection-Expression* [ *Sign-Expression* ]  
    *Element-Selection-Expression* . *Sign-Expression*

*Sign-Expression* :  
    *Primary-Expression*  
    - *Primary-Expression*  
    + *Primary-Expression*

*Primary-Expression* :  
    *Literal-Operand*  
    *Record-Constant*  
    *Identifier-Operand*  
    *Register-Operand*  
    *Integral-TypeName-Operand*

*Value-Substitution-Operand*  
*LENGTH Identifier-Operand*  
*LENGTHOF Identifier-Operand*  
*MASK Identifier-Operand*  
*SIZE Element-Selection-Expression*  
*SIZEOF Element-Selection-Expression*  
*WIDTH Identifier-Operand*  
*Parenthesized-Expression*  
*Indirected-Expression*  
*Compound-Initializer*

*Literal-Operand :*  
*Floating-Point-Literal*  
*Integer-Literal*  
*String-Literal*

*Record-Constant :*  
*Identifier-Operand* < *Field-List* >  
*Identifier-Operand* { *Field-List* }

*Field-List:*  
*Attribute-Expression*  
*Field-List* , *Attribute-Expression*

*Identifier-Operand :*  
*Identifier*

*Register-Operand :*  
*Processor-Register*

*Integral-TypeName-Operand :*  
*Scalar-TypeName*  
*Distance-TypeName*

*Value-Substitution-Operand :*  
*Anonymous-Label-Alias*  
*Location-Counter-Alias*  
*Indeterminate-Value-Alias*  
**FLAT**

*Parenthesized-Expression :*  
( *Attribute-Expression* )

*Indirected-Expression :*  
[ *Attribute-Expression* ]

*Compound-Initializer :*  
< *Initializer-List* >  
{ *Initializer-List* }

*Initializer-List:*  
*Duplicative-Expression*  
*Initializer-List* , *Duplicative-Expression*

---

## Duplicative Initialization Expression

---

### Description

A ***Duplicative Initialization Expression*** is one that can be optionally used during the initialization of variables such that the operand is duplicated a specified number of times.

---

# Syntax

*Duplicative-Expression* :  
    *Attribute-Expression*  
    *Attribute-Expression* DUP ( *Initializer-List* )

*Initializer-List*:  
    *Duplicative-Expression*  
    *Initializer-List* , *Duplicative-Expression*

-----

## Duplicative Initialization (DUP Operator)

-----

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

-----

# Syntax

*Attribute-Expression* DUP ( *Initializer-List* )

*Initializer-List*:  
    *Duplicative-Expression*  
    *Initializer-List* , *Duplicative-Expression*

-----

## Description

The **DUP** operator creates a *Duplicated-ExpressionType* from the *Initializer-List* enclosed in parentheses. This construct can be used to create arrays of information during data allocation.

-----

## Constraints

The left hand operand of the **DUP** operator must evaluate to an *Absolute-ExpressionType* .

Each *Duplicative-Expression* in the *Initializer-List* must evaluate to an *Initializer-ExpressionType* .

---

## Examples

```
STR STRUCT
    One BYTE 0
    Two BYTE 0
STR ENDS

Array1 WORD 4 DUP (1, 2, 3, 4) ; allocates 16 words
Array2 STR 8 DUP (<1, 2>) ; 8 structures
```

---

## Attribute Expression

---

## Description

An ***Attribute Expression*** is one that optionally extracts or modifies one or more of the basic properties of its operand.

---

## Syntax

*Attribute-Expression :*  
    *OR-Expression*  
    SHORT *Additive-Expression*  
    .TYPE *OR-Expression*  
    OPATTR *OR-Expression*

---

## Expression Descriptor Bitmap (.TYPE Operator)

---

## Related Information

[Description](#)

[Examples](#)

---

# Syntax

`.TYPE` *OR-Expression*

---

## Description

The `.TYPE` operator is considered obsolete. The `OPATTR` operator should be used instead.

The `.TYPE` operator returns a byte value bitmap that describes various attributes of its operand. The return value is 0 if the expression could not be correctly parsed or evaluated, otherwise the bitmap returned is formatted according to the following table:

7	6	5	4	3	2	1	0	BIT SET IF EXPRESSION
						1		Is a <i>Direct-ExpressionType</i>
						1		Is a <i>Indirect-ExpressionType</i> , an <i>Indexed-ExpressionType</i> , or a combination of both
						1		Is an <i>Immediate-ExpressionType</i>
						1		Is an <i>Indirect-ExpressionType</i>
						1		Is a <i>Register-ExpressionType</i>
						1		Was parsed and evaluated without error (no undefined symbols, etc.)
						1		Is relative to the <code>SS</code> <i>Segment-Register</i>
						1		Contains an <i>External Reference</i>

---

## Examples

```
BumpCounter macro bump
    if (((.TYPE (bump)) and 07h) eq 04h)
        Counter = Counter + bump
    else
        .err <Non-constant value passed to BumpCounter>
    endif
endm
```

---

## Extended Descriptor Bitmap (OPATTR Operator)

---

# Related Information

- Description
- Constraints
- Examples

## Syntax

**OPATTR** *OR-Expression*

## Description

The **OPATTR** operator returns a superset of the information returned by the **.TYPE** operator, which should be considered obsolete.

The **OPATTR** operator returns a word value bitmap that describes various attributes of its operand. The return value is 0 if the expression could not be correctly parsed or evaluated, otherwise the bitmap returned is formatted according to the following table:

A98	7	6	5	4	3	2	1	0	BIT SET IF EXPRESSION
							1		Is a <i>Direct-ExpressionType</i>
							1		Is a <i>Indirect-ExpressionType</i> , an <i>Indexed-ExpressionType</i> , or a combination of both
							1		Is an <i>Immediate-ExpressionType</i>
							1		Is an <i>Indirect-ExpressionType</i>
							1		Is a <i>Register-ExpressionType</i>
							1		Was parsed and evaluated without error (no undefined symbols, etc.)
							1		Is relative to the <b>SS</b> <i>Segment-Register</i>
							1		Contains an <i>External Reference</i>
LLL									Language encoding (described below)

The **LLL** field (bits 8, 9, and A) comprise an enumerated value that describes the language attribute assigned to the expression as follows:

000	No language attribute used in expression
001	C
010	SYSCALL
011	STDCALL
100	PASCAL
101	FORTRAN
110	BASIC
111	OPTLINK

## Constraints

This operator is not available in [M510](#) mode.

---

## Examples

```
L_MASK      equ 011100000000y          ; mask to isolate language bits
L_OPTLINK   equ 011100000000y          ; setting for OptLink calling convention
VerifyCallBack macro ProcName
    if (((OPATTR (ProcName)) and L_MASK) ne L_OPTLINK)
        .err <Call-back routine must have OptLink linkage>
    endif
endm
```

---

## Force Short Relative Address (SHORT Operator)

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

**SHORT** *Additive-Expression*

---

## Description

The **SHORT** operator forces the assembler to calculate the distance from the start of the next instruction to the target specified by the operand (given by *Additive-Expression*) to be less than 128 bytes away. This can cause the assembler to generate more efficient control transfer instructions when the target is a forward reference. By default, the assembler assumes that the code-relative target is of **NEAR** distance when the target is an unqualified forward reference.

---

## Constraints

The *Additive-Expression* must evaluate to a *Direct-ExpressionType*.

---

## Examples

```
JMP    Forward           ; target unknown, NEAR jump generated
JMP    SHORT Forward     ; force SHORT encoding
.
.           ; fewer than 128 bytes of instructions
.
Forward:           ; definition of target
```

---

## Bitwise OR Expression

## Description

A *Bitwise OR Expression* is one where an optional binary bitwise **OR** operation between the left and right operands is performed and the result returned.

---

## Syntax

```
OR-Expression :
    AND-Expression
    OR-Expression OR AND-Expression
    OR-Expression XOR AND-Expression
```

---

## Bitwise Inclusive OR (OR Operator)

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

*OR-Expression* **OR** *AND-Expression*

---

## Description

The **OR** operator performs a binary bitwise OR operation on the left and right hand operands.

---

## Constraints

Each operand must evaluate to a *Constant-ExpressionType*.

---

## Examples

```
One EQU 1
Two EQU 2

MOV AX, One OR Two ; moves 3 into AX
```

---

## Bitwise Exclusive OR (XOR Operator)

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

# Syntax

*OR-Expression* **XOR** *AND-Expression*

---

## Description

The **XOR** operator performs a binary bitwise XOR operation on the left and right hand operands.

---

## Constraints

Each operand must evaluate to a *Constant-ExpressionType*.

---

## Examples

```
Lower EQU 0101y      ; 7h - binary radix suffix
Upper EQU 1100y      ; Eh - binary radix suffix

MOV AX, Upper XOR Lower ; moves 1001 into AX
```

---

## Bitwise AND Expression

---

## Description

A *Bitwise AND Expression* is one where an optional binary bitwise **AND** operation between the left and right operands is performed and the result returned.

---

## Syntax

*AND-Expression* :  
*NOT-Expression*  
*AND-Expression* **AND** *NOT-Expression*

---

# Bitwise AND (AND Operator)

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

*AND-Expression* **AND** *NOT-Expression*

---

## Description

The **AND** operator performs a binary bitwise AND operation on the left and right hand operands.

---

## Constraints

Each operand must evaluate to a *Constant-ExpressionType*.

---

## Examples

```
Lower EQU 0111y      ; 7h - binary radix suffix
Upper EQU 1110y      ; Eh - binary radix suffix

MOV AX, Upper XOR Lower ; moves 0110 into AX
```

---

## Bitwise One's Complement Expression

---

## Description

A *Bitwise One's Complement Expression* is one that performs an optional unary bitwise negation of its operand and returns the result.

---

## Syntax

*NOT-Expression* :  
    *Relational-Expression*  
    NOT *Relational-Expression*

---

## Bitwise One's Complement (NOT Operator)

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

NOT *Relational-Expression*

---

## Description

The **NOT** operator performs a unary bitwise negation on its operand.

---

## Constraints

The operand must evaluate to a *Constant-ExpressionType*.

---

## Examples

```
Value EQU 0111y          ; 7h - binary radix suffix
MOV EAX, NOT Value        ; moves FFFFFFF8 into EAX
```

---

## Relational Expression

## Description

A **Relational Expression** is one where an optional binary comparison operation between the left and right operands is performed and the result returned.

---

## Syntax

*Relational-Expression :*  
*Additive-Expression*  
*Relational-Expression* EQ *Additive-Expression*  
*Relational-Expression* NE *Additive-Expression*  
*Relational-Expression* GT *Additive-Expression*  
*Relational-Expression* GE *Additive-Expression*  
*Relational-Expression* LT *Additive-Expression*  
*Relational-Expression* LE *Additive-Expression*

---

## Equal To (EQ Operator)

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

-----

## Syntax

*Relational-Expression* **EQ** *Additive-Expression*

-----

## Description

The **EQ** operator performs a binary logical comparison on the left and right hand operands. It returns true (all bits on) if they are equal, and false (all bits off) if they are not equal.

-----

## Constraints

Each operand must evaluate to a *Constant-ExpressionType*.

-----

## Examples

```
IF 1234 EQ 5678
    TRUE = 1
ELSE
    TRUE = 0           ; Sets TRUE to 0
ENDIF
```

-----

## Not Equal To (NE Operator)

-----

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

# Syntax

*Relational-Expression* **NE** *Additive-Expression*

---

## Description

The **NE** operator performs a binary logical comparison on the left and right hand operands. It returns true (all bits on) if they are not equal, and false (all bits off) if they are equal.

---

## Constraints

Each operand must evaluate to a *Constant-ExpressionType*.

---

## Examples

```
IF 1234 NE 5678
    TRUE = 1           ; Sets TRUE to 1
ELSE
    TRUE = 0
ENDIF
```

---

## Greater Than (GT Operator)

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

-----

## Description

The **GT** operator performs a binary logical comparison on the left and right hand operands. It returns true (all bits on) if the left operand is greater than the right operand, and false (all bits off) if it is not.

-----

## Constraints

Each operand must evaluate to a *Constant-ExpressionType*.

-----

## Examples

```
IF 1234 GT 5678
    TRUE = 1
ELSE
    TRUE = 0          ; Sets TRUE to 0
ENDIF
```

-----

## Greater Than or Equal To (GE Operator)

-----

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

-----

## Syntax

---

## Description

The **GE** operator performs a binary logical comparison on the left and right hand operands. It returns true (all bits on) if the left operand is greater than or equal to the right operand, and false (all bits off) if it is not.

---

## Constraints

Each operand must evaluate to a *Constant-ExpressionType*.

---

## Examples

```
IF 1234 GE 1234
    TRUE = 1          ; Sets TRUE to 1
ELSE
    TRUE = 0
ENDIF
```

---

## Less Than (LT Operator)

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

*Relational-Expression* LT *Additive-Expression*

---

# Description

The **LT** operator performs a binary logical comparison on the left and right hand operands. It returns true (all bits on) if the left operand is less than the right operand, and false (all bits off) if it is not.

-----

# Constraints

Each operand must evaluate to a *Constant-ExpressionType*.

-----

# Examples

```
IF 1234 LT 5678
    TRUE = 1          ; Sets TRUE to 1
ELSE
    TRUE = 0
ENDIF
```

-----

# Less Than or Equal To (LE Operator)

-----

# Related Information

[Description](#)

[Constraints](#)

[Examples](#)

-----

# Syntax

*Relational-Expression* **LE** *Additive-Expression*

-----

# Description

The **LE** operator performs a binary logical comparison on the left and right hand operands. It returns true (all bits on) if the left operand is less than or equal to the right operand, and false (all bits off) if it is not.

-----

## Constraints

Each operand must evaluate to a *Constant-ExpressionType*.

-----

## Examples

```
IF 1234 LE 1234
    TRUE = 1          ; Sets TRUE to 1
ELSE
    TRUE = 0
ENDIF
```

-----

## Additive Expression

-----

## Description

A ***Additive Expression*** is one where an optional binary additive arithmetic operation between the left and right operands is performed and the result returned.

-----

## Syntax

*Additive-Expression* :

- Multiplicative-Expression*
- Additive-Expression* + *Multiplicative-Expression*
- Additive-Expression* - *Multiplicative-Expression*

-----

## Addition (+ Operator)

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

*Additive-Expression* + *Multiplicative-Expression*

---

## Description

The + operator performs a binary addition operation on the left and right hand operands, and returns the result.

---

## Constraints

One of the operands must evaluate to a *Constant-ExpressionType*. If one of the operands references an external identifier, then the other operand must be a *Constant-ExpressionType* without an external reference. Both operands must be of scalar type.

---

## Examples

```
VALUE = 100 + 11          ; sets VALUE to 111
```

---

## Subtraction (- Operator)

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

-----

## Syntax

*Additive-Expression* - *Multiplicative-Expression*

-----

## Description

The - operator performs a binary subtraction operation on the left and right hand operands, and returns the result.

-----

## Constraints

The right operand must evaluate to a *Constant-ExpressionType* and reference no external identifiers. If both operands are relocatable, they must reside within the same segment, in which case the result is converted to a *Absolute-ExpressionType*. Both operands must be of scalar type.

-----

## Examples

```
VALUE = 111 - 11      ; sets VALUE to 100
```

-----

## Multiplicative Expression

-----

## Description

A *Multiplicative Expression* is one where an optional binary multiplicative arithmetic operation between the left and right operands is performed and the result returned.

-----

## Syntax

*Multiplicative-Expression :*  
*Narrowed-Expression*  
*Multiplicative-Expression* \* *Narrowed-Expression*  
*Multiplicative-Expression* | *Narrowed-Expression*  
*Multiplicative-Expression* MOD *Narrowed-Expression*  
*Multiplicative-Expression* SHL *Narrowed-Expression*  
*Multiplicative-Expression* SHR *Narrowed-Expression*

---

## Multiplication (\* Operator)

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

*Multiplicative-Expression* \* *Narrowed-Expression*

---

## Description

The \* operator performs a binary multiplication operation on the left and right hand operands, and returns the result.

---

## Constraints

Each operand must evaluate to a *Constant-ExpressionType*.

---

## Examples

```
VALUE = 9 * 3          ; sets VALUE to 27
```

---

## Division (/ Operator)

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

*Multiplicative-Expression* / *Narrowed-Expression*

---

## Description

The / operator performs a binary division operation on the left and right hand operands, and returns the result.

---

## Constraints

Each operand must evaluate to a *Constant-ExpressionType*.

---

## Examples

```
VALUE = 27 / 9          ; sets VALUE to 3
```

---

## Remainder (MOD Operator)

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

*Multiplicative-Expression* **MOD** *Narrowed-Expression*

---

## Description

The **MOD** operator performs a binary modulus division operation on the left and right hand operands, and returns the remainder as the result.

---

## Constraints

Each operand must evaluate to a *Constant-ExpressionType*.

---

## Examples

```
VALUE = 18 MOD 4      ; sets VALUE to 2
```

---

## Bitwise Left Shift (SHL Operator)

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

-----

## Syntax

*Multiplicative-Expression* **SHL** *Narrowed-Expression*

-----

## Description

The **SHL** operator shifts the bits in the left hand operand to the left by the number of bits specified in the right hand operand, and returns the result.

-----

## Constraints

Each operand must evaluate to a *Constant-ExpressionType*.

-----

## Examples

```
VALUE = 1111y SHL 4      ; sets VALUE to 11110000y
```

-----

## Bitwise Right Shift (SHR Operator)

-----

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

*Multiplicative-Expression* **SHR** *Narrowed-Expression*

---

## Description

The **SHR** operator shifts the bits in the left hand operand to the right by the number of bits specified in the right hand operand, and returns the result.

---

## Constraints

Each operand must evaluate to a *Constant-ExpressionType*.

---

## Examples

```
VALUE = 11110000y SHR 4 ; sets VALUE to 00001111y
```

---

## Narrowed Expression

---

## Description

A *Narrowed Expression* is one that performs an optional unary narrowing operation on its operand and returns the result.

---

## Syntax

*Narrowed-Expression* :  
    *Cast-Expression*  
    HIGH *Cast-Expression*  
    HIGHWORD *Cast-Expression*

---

## Upper 8 Bits of WORD Expression (HIGH Operator)

---

### Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

### Syntax

**HIGH** *Cast-Expression*

---

### Description

The **HIGH** operator returns the upper 8 bits of a 16-bit expression. Only bits 8-15 are returned, even if the magnitude of the operand exceeds 16 bits.

---

### Constraints

The operand must evaluate to a *Constant-ExpressionType*.

---

### Examples

```
FIRST  = 1234h  
SECOND = HIGH FIRST      ; Sets SECOND to 12h
```

# Upper 16 Bits of DWORD Expression (HIGHWORD Operator)

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

**HIGHWORD** *Cast-Expression*

---

## Description

The **HIGHWORD** operator returns the upper 16 bits of a 32-bit expression. Only bits 16-31 are returned, even if the magnitude of the operand exceeds 32 bits.

---

## Constraints

The operand must evaluate to a *Constant-ExpressionType*.

This operator is not available in **M510** mode.

---

## Examples

```
FIRST  = 12345678h
SECOND = HIGHWORD FIRST    ; Sets SECOND to 1234h
```

---

## Lower 8 Bits of WORD Expression (LOW Operator)

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

**LOW** *Cast-Expression*

---

## Description

The **LOW** operator returns the lower 8 bits of its operand.

---

## Constraints

The operand must evaluate to a *Constant-Expression Type*.

---

## Examples

```
FIRST  = 1234h
SECOND = LOW FIRST      ; Sets SECOND to 34h
```

---

## Lower 16 Bits of DWORD Expression (LOWWORD Operator)

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

**LOWWORD** *Cast-Expression*

---

## Description

The **LOWWORD** operator returns the lower 16 bits of its operand.

---

## Constraints

The operand must evaluate to a *Constant-Expression Type*.

This operator is not available in **M510** mode.

---

## Examples

```
FIRST  = 12345678h
SECOND = LOWWORD FIRST    ; Sets SECOND to 5678h
```

---

## Type Conversion Expression

---

## Description

A *Type Conversion Expression* is one that performs an optional type conversion operation on its operand and returns the result.

---

## Syntax

*Cast-Expression* :  
    *Element-Selection-Expression*  
    **OFFSET** *Cast-Expression*  
    **SEG** *Cast-Expression*  
    **THIS** *Element-Selection-Expression*  
    **TYPE** *Element-Selection-Expression*  
    *Cast-Expression* **PTR** *Cast-Expression*  
    *Cast-Expression* : *Cast-Expression*

---

# Address Offset (OFFSET Operator)

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

**OFFSET** *Cast-Expression*

---

## Description

The **OFFSET** operator returns the offset portion of its operand. For relocatable values, this is the offset into the segment or group to which the expression is relative.

---

## Constraints

The operand may evaluate to any one of the following *ExpressionType*s :

- *Absolute-ExpressionType*
  - *Constant-ExpressionType*
  - *Immediate-ExpressionType*
  - *Direct-ExpressionType*
  - *Indirect-ExpressionType*
-

# Examples

```
CodeLabel:
    MOV AX,CodeLabel      ; illegal, no data at address
    MOV AX,OFFSET CodeLabel ; we want the address itself
```

---

## Address Segment (SEG Operator)

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

## Syntax

**SEG** *Cast-Expression*

---

## Description

The **SEG** operator returns the segment or group to which a relocatable expression is relative.

---

## Constraints

The operand must evaluate to one of the following *ExpressionType*s:

- *Immediate-ExpressionType*
  - *Direct-ExpressionType*
  - *Indirect-ExpressionType*
  - *Indexed-ExpressionType*
-

# Examples

```
DATA    SEGMENT
Stuff  DB    ?
        MOV  AX, SEG Stuff  ; This construct is
        MOV  AX, DATA      ; equivalent to this
DATA    ENDS
```

## Address Alias (THIS Operator)

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

## Syntax

**THIS** *Element-Selection-Expression*

## Description

The **THIS** operator returns an operand whose:

- *Relative Frame* attribute is set to that of the current segment
- *Displacement* attribute is set to the current location counter
- *Type Declaration* attribute is set to that of the expression given by the *Element-Selection-Expression* operand.

## Constraints

The operand must evaluate to a *Type-ExpressionType*.

# Examples

```
DATA      SEGMENT

ALIAS     EQU    THIS BYTE      ; reference this address as a byte
Stuff     DB     ?

          MOV     AL, ALIAS      ; This construct is
          MOV     AL, Stuff      ; equivalent to this

DATA      ENDS
```

## Datatype Extraction (TYPE Operator)

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

## Syntax

**TYPE** *Element-Selection-Expression*

## Description

The **TYPE** operator returns the *Type-ExpressionType* attribute of its operand.

## Constraints

None

## Examples

```
CODE    SEGMENT
        ASSUME CS:CODE,DS:CODE
Stuff   DB    ?                ; TYPE Stuff is BYTE
        MOV   [BX], (TYPE Stuff) PTR 1    ; stores 1 as a BYTE at [BX]
CODE    ENDS
```

---

## Type Conversion (PTR Operator)

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

*Cast-Expression* **PTR** *Cast-Expression*

---

## Description

The **PTR** operator converts the right operand to the type specified by the left operand.

---

## Constraints

The left operand must be a *Type-ExpressionType*.

---

## Examples

```
CODE      SEGMENT
MOV      BYTE PTR [BX], 1      ; stores 1 as a BYTE at [BX]
CODE      ENDS
```

-----

## Segment Override (: Operator)

-----

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

-----

## Syntax

*Cast-Expression* : *Cast-Expression*

-----

## Description

The : (colon) operator forces the right operand to have the *Relative Frame* attribute of the left operand.

-----

## Constraints

The left operand must evaluate to one of the following *ExpressionType*s :

- *Register-ExpressionType* where the *Register Value* attribute is that of a *Segment-Register*
- *Immediate-ExpressionType* where the *Relative Frame* attribute is that of a *GroupName* or *SegmentName*.

-----

## Examples

```
DATA      SEGMENT
Variable DW      ?
DATA      ENDS
```

```

DGROUP    GROUP DATA, CODE

CODE      SEGMENT
  ASSUME  CS:CODE, DS:DGROUP
  MOV     AX, DGROUP:Variable      ; insure Variable is relative to DGROUP
  ASSUME  DS:NOTHING
  MOV     BX, CS:Variable          ; access Variable through CS register
CODE      ENDS

```

## Element Selection Expression

## Description

A *Element Selection Expression* is one that optionally selects a specific element of its operand and returns a reference to it.

## Syntax

```

Element-Selection-Expression :
  Sign-Expression
  Element-Selection-Expression [ Sign-Expression ]
  Element-Selection-Expression . Sign-Expression

```

## Subscript ([] Operator)

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

## Syntax

```

Element-Selection-Expression [ Sign-Expression ]

```

---

## Description

The `[]` binary operator performs a subscripting (or *indexing*) operation between the operand to the left of the brackets and the operand enclosed within the brackets. This is a simple additive operation of **BYTE** granularity; the arithmetic performed is not influenced by the *Operand Size* of either operand.

The syntax for this operator describes a binary operation between the left hand expression and the bracketed expression. The bracketed expression is also subject to the same operations performed during the processing of a standalone *Indirected-Expression* as described in the section on *Primary-Expression*s.

---

## Constraints

Only one of the operands may specify a relocatable value.

---

## Examples

```
CODE      SEGMENT
          ASSUME CS:CODE, DS:CODE

Value     DB  0                ; Value[0]
          DB  1                ; Value[1]
          DB  2                ; Value[2]
          DB  3                ; Value[3]
          DB  4                ; Value[4]

          MOV  AL, Value[3]      ; load AL with the fourth byte at Value (3)
          MOV  BX, offset Value ; get address of Value
          MOV  AL, [BX][1][2]    ; also gets the fourth byte (3)

CODE      ENDS
```

---

## Structure/Union Field Selection (. Operator)

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

# Syntax

*Element-Selection-Expression* . *Sign-Expression*

---

## Description

The . (period) operator selects a structure or union field entry. It adds the left and right hand operands together and returns the result. The left operand should be an *Indirect-ExpressionType*, *Indexed-ExpressionType*, or *Type-ExpressionType* whose *Type Declaration* attribute resolves to that of a *Structure-TypeName* or *Union-TypeName*. The right operand should refer to a *FieldName* defined within the referenced type.

The *Operand Size* attribute of the result depends on the operands involved. If both operands have an operand size, a *Structure-FieldName* appearing as the right hand operand would override the operand size of the left operand and would dictate the operand size of the resulting expression.

---

## Constraints

Only one of the operands may specify a relocatable value.

---

## Examples

```
Number STRUC
    One  DB 1
    Two  DW 2
Number ENDS

; The following line is only allowed in MASM 5.10 mode (OPTION OLDSTRUCTS)
MOV AX,[BX].Two      ; BX points to a "Number", get the "Two" entry

; In other modes, "Two" is private to the "Number" structure type, so
; one of the following methods are required:

MOV AX,(Number PTR [BX]).Two      ; Explicit override
MOV AX,[BX] + Number.Two          ; Fully qualified reference
ASSUME BX:Number                  ; Associate BX with "Number"
MOV AX,[BX].Two                   ; then original syntax is allowed
```

---

## Unary Arithmetic Expression

---

## Description

A *Unary Arithmetic Expression* is one that optionally alters the sign of its operand and returns the result.

---

## Syntax

*Sign-Expression* :

- Primary-Expression*
- *Primary-Expression*
- + *Primary-Expression*

---

## Unary Minus (- Operator)

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

- *Primary-Expression*

---

## Description

The - operator makes its operand into a negative number and returns the result.

---

## Constraints

The operand must evaluate to a *Constant-ExpressionType*.

---

## Examples

```
Value EQU 1
MOV AX,-Value ; move -1 into AX
```

---

## Unary Plus (+ Operator)

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

`+ Primary-Expression`

---

## Description

The `+` operator returns its operand.

---

## Constraints

The operand must evaluate to a *Constant-ExpressionType*.

---

## Examples

```
Value EQU 1
MOV AX, +Value ; move 1 into AX
```

---

## Primary Expression

---

## Description

A *Primary Expression* is one that returns an expression operand.

---

## Syntax

*Primary-Expression :*

- Literal-Operand*
- Record-Constant*
- Identifier-Operand*
- Register-Operand*
- Integral-TypeName-Operand*
- Value-Substitution-Operand*
- LENGTH Identifier-Operand*
- LENGTHOF Identifier-Operand*
- MASK Identifier-Operand*
- SIZE Element-Selection-Expression*
- SIZEOF Element-Selection-Expression*
- WIDTH Identifier-Operand*
- Parenthesized-Expression*
- Indirected-Expression*
- Compound-Initializer*

---

## Literal Operand

---

## Related Information

[Description](#)

[Constraints](#)

---

## Syntax

*Literal-Operand:*  
*Floating-Point-Literal*  
*Integer-Literal*  
*String-Literal*

-----

## Description

The assembler accepts several types of literal values as operands within expressions. *Literal-Operand*s are converted to *ExpressionType*s according to the following table:

<i>Floating-Point-Literal</i>	<i>Floating-Point-ExpressionType</i>
<i>Integer-Literal</i>	<i>Absolute-ExpressionType</i>
<i>String-Literal</i>	<i>Absolute-ExpressionType</i> if the string length is less than or equal to the current <i>Address Size</i> ; a <i>String-ExpressionType</i> otherwise.

The context where the expression is used determines whether or not a particular type of literal is legal.

-----

## Constraints

Arithmetic operations cannot be performed on *Floating-Point-Literal*s, thus they cannot be the operand of a unary or binary operator.

-----

## Value Substitution Operand

-----

## Related Information

[Description](#)

[Constraints](#)

-----

## Syntax

*Value-Substitution-Operand:*  
*Anonymous-Label-Alias*  
*Location-Counter-Alias*  
*Indeterminate-Value-Alias*  
FLAT

---

## Description

These operands are used to retrieve specialized values that are calculated internally by the assembler.

The **FLAT** operator returns an expression whose *Relative Frame* is set to that of the predefined FLAT pseudo-group.

---

## Constraints

The **FLAT** operand is only active when a 32-bit processor has been selected.

---

## Record Constant Operand

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

*Record-Constant* :  
    *Identifier-Operand* < *Field-List* >  
    *Identifier-Operand* { *Field-List* }

*Field-List* :  
    *Attribute-Expression*  
    *Field-List* , *Attribute-Expression*

---

## Description

A *Record-Constant* provides a method of calculating a single numeric result value from a list of *Record-FieldName* values, and combining them together according to the definition of the *Record-TypeName* given by the *Identifier-Operand*. The result value is a *Constant-ExpressionType* suitable for use as an instruction operand, or for assigning to a record variable.

The *Record-TypeName* given by the *Identifier-Operand* determines how the *Field-List* will be evaluated. The *Attribute-Expression* entries

are position-dependent, and are matched with the corresponding *Record-FieldName* entries from the *Record-TypeName* definition to determine their width and shift values. *Attribute-Expression* entries may be omitted, in which case the default values from the record definition are used in the calculation.

---

## Constraints

The *Identifier-Operand* must resolve to a *Record-TypeName*.

---

## Examples

```
DATE_T record Year :7 = 0, ; 0 is 1980
              Month:4 = 1, ; January
              Day :5 = 1 ; 1st

CODE SEGMENT
    mov AX, DATE_T <> ; January 1st, 1980
    mov AX, DATE_T <1996-1980, 12, 25> ; Christmas, 1996
    mov AX, DATE_T <10h, 0Ch, 19h> ; equivalent values in hex
    mov AX, DATE_T <10000y, 1100y, 11001y> ; equivalent values in binary
    mov AX, 2199h ; equivalent value manually coded
    mov AX, 0010000110011001y ; and in binary
; YYYYYYYYMMMMDDDDD
CODE ENDS
```

---

## Register Operand

---

## Related Information

[Description](#)

[Constraints](#)

---

## Syntax

*Register-Operand* :  
*Processor-Register*

---

## Description

Processor registers are valid expression operands. The context where the expression is used determines the allowable register operands.

## Constraints

The currently selected processor dictates whether or not a register is visible to the expression evaluator.

## Identifier Operand

## Related Information

[Description](#)

[Constraints](#)

## Syntax

*Identifier-Operand:*  
*Identifier*

## Description

When an *Identifier* is used in an expression, it returns a value according to its *Identifier-Type*, as shown in the following table:

<i>Identifier-Type</i>	VALUE RETURNED
<i>Numeric-EquateName</i>	The value originally assigned to the equate.
<i>Structure-FieldName</i>	The offset in bytes from the beginning of the structure.
<i>Union-FieldName</i>	The offset in bytes from the beginning of the union (always 0).
<i>Record-FieldName</i>	The shift-count required to reach the field within the record.
<i>Record-TypeName</i>	The mask-value that isolates defined record fields from undefined fields.
<i>Structure-TypeName</i>	Zero if mode is <a href="#">M510</a> , otherwise the size of the structure in bytes (the operand size of the structure type).
<i>Union-TypeName</i>	The size of the union in bytes (the operand size of the union type).

<i>Typedef-TypeName</i>	The operand size of the underlying data-type represented by the <i>Typedef-TypeName</i> .
<i>GroupName</i>	A <i>Relative Frame</i> attribute that represents the group, and a <i>Displacement</i> value of zero.
<i>SegmentName</i>	A <i>Relative Frame</i> attribute that represents the segment (or the group to which it belongs), and a <i>Displacement</i> value of zero if the mode is <i>M510</i> , or the current segment offset otherwise.
<i>LabelName</i>	The <i>Relative Frame</i> attribute where the label is defined, and the segment offset value of the label.

## Constraints

The *Identifier* must resolve to one of the following *Identifier-Type* s:

- *Numeric-EquateName*
- *FieldName*
- *GroupName*
- *LabelName*
- *SegmentName*
- *UserDefined-TypeName*

## Integral Type-Name Operand

## Related Information

Description

Constraints

## Syntax

```
Integral-TypeName-Operand:
    Scalar-TypeName
    Distance-TypeName
```

## Description

When an *Integral-TypeName-Operand* is used in an expression, it is converted to a *Type-ExpressionType*. If used in a numeric context, the

following numeric values are returned:

*Integral-TypeName-Operand* VALUE RETURNED

*Scalar-TypeName* The operand-size of the type in bytes.

*Distance-TypeName* If mode is **M510**, NEAR returns FFFF, and FAR returns FFFE. Otherwise, NEAR and FAR are resolved and the values returned are: NEAR16=FF02, NEAR32=FF04, FAR16=FF05, FAR32=FF06.

---

## Constraints

The **NEAR32** and **FAR32** keywords are only valid if a 32-bit processor has been selected.

---

## Number of Data Elements (LENGTH Operator)

---

## Related Information

[Description](#)

[Constraints](#)

---

## Syntax

**LENGTH** *Identifier-Operand*

---

## Description

The **LENGTH** operator returns the number of data elements allocated to the operand. When applied to a variable initialized with a series of comma-separated expressions (elements), only the length of the first element is considered.

---

## Constraints

The operand must evaluate to a *Data-Label/Name*.

---

## Number of Data Elements (LENGTHOF Operator)

---

### Related Information

[Description](#)

[Constraints](#)

---

### Syntax

**LENGTHOF** *Identifier-Operand*

---

### Description

The **LENGTHOF** operator returns the number of data elements allocated to the operand.

---

### Constraints

The operand must evaluate to a *Data-Label/Name*.

This operator is not available in **M510** mode.

---

### Examples

<none>

---

## Record or Field Bit-Mask (MASK Operator)

---

### Related Information

[Description](#)

[Constraints](#)

-----

## Syntax

**MASK** *Identifier-Operand*

-----

## Description

The **MASK** operator returns the bit mask required to isolate a field within a record.

-----

## Constraints

The *Identifier-Operand* must resolve to a *Record-TypeName* or *Record-FieldName*; otherwise the result is zero.

-----

## Size of Variable in Bytes (SIZE Operator)

-----

## Related Information

[Description](#)

[Constraints](#)

-----

## Syntax

**SIZE** *Element-Selection-Expression*

-----

## Description

The **SIZE** operator returns the number of bytes allocated to the operand. When applied to a variable initialized with a series of

comma-separated expressions (elements), only the size of the first element is considered.

---

## Constraints

None

---

## Size of Variable in Bytes (SIZEOF Operator)

## Related Information

[Description](#)

[Constraints](#)

---

## Syntax

**SIZEOF** *Element-Selection-Expression*

---

## Description

The **SIZEOF** operator returns the number of bytes allocated to the operand.

---

## Constraints

This operator is not available in [M510](#) mode.

---

## Record or Field Width (WIDTH Operator)

## Related Information

Description

Constraints

-----

## Syntax

**WIDTH** *Identifier-Operand*

-----

## Description

The **WIDTH** operator returns the width of a record or a record field name.

-----

## Constraints

The *Identifier-Operand* must resolve to a *Record-TypeName* or *Record-FieldName*; otherwise the result is zero.

-----

## Precedence (()) Operator

-----

## Related Information

Description

Examples

-----

## Syntax

*Parenthesized-Expression* :  
( *Attribute-Expression* )

-----

## Description

Parentheses forces the *Attribute-Expression* operand to be evaluated at a higher precedence level.

---

## Examples

```
Value = 2 + 3 * 4      ; Value = 14
Value = (2 + 3) * 4    ; Value = 20
```

---

## Indirection ([] Operator)

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

*Indirected-Expression* :  
[ *Attribute-Expression* ]

---

## Description

During evaluation of the *Attribute-Expression*, the [] (*indirection*) operator will convert a *Register-ExpressionType* to a *Indexed-ExpressionType* by moving the *Register Value* attribute to either the *Base Register* or *Index Register* attribute field as appropriate for the register(s) referenced in the expression. This operation allows values contained in the processor registers to be used during effective address calculation at application run time.

---

## Constraints

See the *Indexed-ExpressionType* section for information on registers that are valid for use in this context.

---

# Examples

```
CODE      SEGMENT
          ASSUME CS:CODE, DS:CODE
Value     DW      0
          MOV     BX,offset Value      ; load the address of Value into BX
          MOV     [BX],BX             ; store the contents of BX into the
                                     ; memory location addressed by [BX]
CODE      ENDS
```

## Compound Initializer List ( Operator)

## Related Information

[Description](#)

[Examples](#)

## Syntax

```
Compound-Initializer:
    < Initializer-List >
    { Initializer-List }

Initializer-List:
    Duplicative-Expression
    Initializer-List , Duplicative-Expression
```

## Description

The <> (or {}) operator provides a way of specifying a list of expressions to be used for initializing complex (multi-field) variables such as records or structures.

The <> operator encloses a list of comma-separated expressions; individual expressions are optional, but are also positional with respect to the record or structure fields they are intended to initialize. Commas must therefore be used to maintain field positions if empty expressions are encountered in the list.

The initializer list itself may also be left out entirely for those cases where a variable allocation will use the default initializers provided in the record or structure definition (the <> or {} themselves are still required).

# Examples

```
Numbers STRUCT
    One    DB    0
    Two    DW    0
    Three  DB    0
    Four   DD    0
Numbers ENDS

First  Numbers <>          ; empty initializer list
Second Numbers <1, 2, 3, 4> ; override all defaults
Third  Numbers <1>         ; override first entry only
Fourth Numbers <1,,4>      ; override first and last entries
```

---

## Expression Evaluation

After an expression is parsed and checked for syntax errors, it is *evaluated*. During evaluation, all calculations and conversions are performed on the operands according to the operators that are applied to them. The final result is a collection of *Expression-Attribute* *s*, to which an *ExpressionType* is assigned.

---

## Expression Attributes

This section describes the *Expression-Attribute* *s* that are associated with an expression after it is evaluated.

---

## Address Size

If an expression refers to an effective address, then it also has an associated *address size*. The following *ExpressionType* *s* normally reference an effective address, and thus have an associated address size:

- *Immediate-ExpressionType*
- *Direct-ExpressionType*
- *Indirect-ExpressionType*
- *Indexed-ExpressionType*

The address size can be either 2 (**USE16**) or 4 (**USE32**). For an expression that references a label, the address size of the segment where the label is defined determines the address size of the expression.

---

## Operand Size

The *Operand Size* of an expression can be set explicitly using the *Type Conversion (PTR Operator)*, or it may be a side-effect inherited from the type of data referenced in the expression. The following table describes the operand sizes that will be assigned when an identifier is referenced in an expression:

REFERENCE	OPERAND SIZE
<i>8-Bit-Register</i>	1

<i>16-Bit-Register</i>	2
<i>32-Bit-Register</i>	4
<i>Segment-Register</i>	2
<i>Control-Register</i>	4
<i>Debug-Register</i>	4
<i>Test-Register</i>	4
<i>MMX-Register</i>	8
<i>Floating-Point-Register</i>	10
BYTE	1
SBYTE	1
WORD	2
SWORD	2
DWORD	4
SDWORD	4
REAL4	4
FWORD	6
QWORD	8
REAL8	8
TBYTE	10
REAL10	10
NEAR	2 or 4
NEAR16	2
NEAR32	4
FAR	4 or 6
FAR16	4
FAR32	6
<i>Numeric-EquateName</i>	Inherited from equate expression
<i>GroupName</i>	2
<i>SegmentName</i>	2
<i>Code-LabelName</i>	SIZE (TYPE <i>Code-LabelName</i> )
<i>Data-LabelName</i>	SIZE (TYPE <i>Data-LabelName</i> )
<i>Structure-FieldName</i>	SIZE <i>Structure-FieldName</i>
<i>Record-TypeName</i>	SIZE <i>Record-TypeName</i>
<i>Structure-TypeName</i>	SIZE <i>Structure-TypeName</i>
<i>Union-TypeName</i>	SIZE <i>Union-TypeName</i>

The *Operand Size* is 0 for all other identifier types.

-----

## Displacement

The *Displacement* value in an expression is the final calculated value of all numeric quantities, and must be a scalar value. It may also be a

reference to a relocatable address, in which case the expression will also have a *Relative Frame* and/or an *External Reference* attribute. A *Displacement* may be used in the calculation of an effective address, either alone or in combination with a *Base Register* and/or an *Index Register*.

-----

## Relative Frame

The *Relative Frame* attribute will be present if the expression contains a direct or indirect reference to any of the following *Identifier-Type*s :

- *GroupName*
- *LabelName*
- *SegmentName*

The *Relative Frame* attribute indicates that the expression is relocatable, and specifies the *GroupName* or *SegmentName* to which the expression is relative.

-----

## External Reference

The *External Reference* attribute will be present if the expression references any external identifiers.

-----

## Register Value

The *Register Value* attribute specifies the value of the *Processor-Register* referenced in a *Register-ExpressionType*.

-----

## Base Register

The *Base Register* attribute specifies the value for the base register used in an *Indexed-ExpressionType*.

-----

## Index Register

The *Index Register* attribute specifies the value for the index register used in an *Indexed-ExpressionType*.

-----

## Scale Factor

The *Scale Factor* attribute specifies the scaling value used (if any) in an *Indexed-ExpressionType*.

-----

## Type Declaration

The *Type Declaration* attribute specifies the type of data referenced in the expression. This is the value extracted from the expression when it is used as the left operand of the *Type Conversion (PTR Operator)*.

---

# Expression Types

---

## Description

An *ExpressionType* is assigned to every expression during evaluation. The *ExpressionType* is used to determine whether or not an expression is legal for the context in which it is used. The type of an expression is influenced primarily by the operands that are used, but the use of expression operators also play an important part in determining the type of an expression.

---

## Definition

*ExpressionType* :

- Absolute-ExpressionType*
- Constant-ExpressionType*
- Direct-ExpressionType*
- Floating-Point-ExpressionType*
- Immediate-ExpressionType*
- Indirect-ExpressionType*
- Indexed-ExpressionType*
- Register-ExpressionType*
- String-ExpressionType*
- Type-ExpressionType*
- Duplicated-ExpressionType*
- Compound-ExpressionType*

---

## Absolute Expression Type

An *Absolute-ExpressionType* is an expression that evaluates to an integer quantity. Its value must be representable using one of the following types of scalar data:

- **BYTE**
- **SBYTE**
- **WORD**
- **SWORD**
- **DWORD**
- **SDWORD**
- **FWORD**
- **QWORD**
- **TBYTE**

The following restrictions apply to an *Absolute-ExpressionType* :

- It cannot be relocatable (it may not contain references to a *GroupName*, *SegmentName* or *LabelName*).
  - It cannot reference any **external** symbols.
  - It cannot contain any forward references.
- 

## Constant Expression Type

A *Constant-ExpressionType* is an *Absolute-ExpressionType* with the following restrictions relaxed:

- It may contain forward references to identifiers defined later in the source stream.
- It may reference a single *external* symbol, provided that the symbol was declared in an **EXTERN** directive with the **ABS** attribute.

---

## Immediate Expression Type

An *Immediate-ExpressionType* has all the properties of a *Constant-ExpressionType* with the following restrictions relaxed:

- It may contain references to a *GroupName*, *SegmentName* or *LabelName* (it may be relocatable).
- It may reference a relocatable *external* symbol.

An *Immediate-ExpressionType* must not be larger than 32 bits in magnitude; its value must be representable using one of the following types of scalar data:

- **BYTE**
- **SBYTE**
- **WORD**
- **SWORD**
- **DWORD**
- **SDWORD**

---

## Direct Expression Type

A *Direct-ExpressionType* is an expression that references a *Code-LabelName*. It can be used directly in code-relative instructions without conversion. There is no data type associated with the address that a *Direct-ExpressionType* represents, therefore It may not be used in a data-relative instruction without first being explicitly converted to another expression type.

---

## Indirect Expression Type

An *Indirect-ExpressionType* is an expression that references a *Data-LabelName*. It can be used directly in data-relative instructions without conversion to another expression type.

---

## Indexed Expression Type

An *Indexed-ExpressionType* is an expression that calculates an effective memory address using the contents of a *Base-Register*, an *Index-Register*, or both. A *Processor-Register* must first be converted to a *Base-Register* or *Index-Register* by specifying it as the operand of the *Indirection* (**[ ] Operator**) before the expression can be converted to an *Indexed-ExpressionType*.

When calculating a 16-bit effective address, only the **BP** and **BX** registers may be used as *Base-Registers*, and only the **DI** and **SI** registers may be used as *Index-Registers*.

When calculating a 32-bit effective address, only the **EAX**, **EBX**, **ECX**, **EDX**, **EDI**, **ESI**, **EBP**, and **ESP** registers may be used as *Base-Registers*, and only the **EAX**, **EBX**, **ECX**, **EDX**, **EDI**, **ESI**, and **EBP** registers may be used as *Index-Registers*.

**Note:** Only a single *Base-Register* and a single *Index-Register* may be used in a given expression.

On 80386 (and higher) processors, the *Multiplication* (**\* Operator**) may be used with an *Index-Register* operand and an *Absolute-ExpressionType* operand to establish a scaling factor that is applied to the *Index-Register* during effective address calculation. The scaling factor effectively causes the *Index-Register* to be multiplied by a fixed value at run time. The scaling *Expression* must evaluate to 1

(no scale factor), 2, 4, or 8.

A *Direct-ExpressionType* or an *Indirect-ExpressionType* may be a sub-expression of an *Indexed-ExpressionType*.

---

## Register Expression Type

A *Register-ExpressionType* is an expression that specifies a single *Processor-Register*.

---

## String Expression Type

A *String-ExpressionType* is an expression that specifies a single *String-Literal*.

---

## Floating-Point Expression Type

A *Floating-Point-ExpressionType* is an expression that specifies a single *Floating-Point-Literal*.

---

## Type Expression Type

A *Type-ExpressionType* is an expression that specifies one of the following:

- A *Scalar-TypeName*
  - A *Distance-TypeName*
  - A *UserDefined-TypeName*
- 

## Compound Expression Type

A *Compound-ExpressionType* evaluates to a list of (possibly nested) expressions collected together as a unit by the *Compound Initializer List (<> Operator)*. A *Compound-ExpressionType* is used to initialize *aggregate* data types (such as records, structures, and unions) and *vector* data types (arrays).

---

## Duplicated Expression Type

A *Duplicated-ExpressionType* evaluates to an expression that is to be duplicated (repeated) a specified number of times. This type of expression is created using the *Duplicative Initialization (DUP Operator)*.

---

## Operand Expression Type

---

## Description

An *Operand-ExpressionType* consists of those *ExpressionType* *s* that are valid for use as operands in processor instructions. The following *ExpressionType* *s* are not valid for use as an *Operand-ExpressionType* :

- *Compound-ExpressionType*
- *Duplicated-ExpressionType*

A *String-ExpressionType* is only valid as an *Operand-ExpressionType* if it is short enough to be converted to an *Absolute-ExpressionType* having an *Operand Size* less than or equal to the current *Address Size* setting.

---

## Definition

*Operand-ExpressionType* :

*Absolute-ExpressionType*  
*Constant-ExpressionType*  
*Immediate-ExpressionType*  
*Direct-ExpressionType*  
*Indirect-ExpressionType*  
*Indexed-ExpressionType*  
*Register-ExpressionType*  
*String-ExpressionType*  
*Floating-Point-ExpressionType*  
*Type-ExpressionType*

---

## Initializer Expression Type

---

## Description

An *Initializer-ExpressionType* consists of those *ExpressionType* *s* that are valid for use in initializing variables. The following *ExpressionType* *s* are not valid *Initializer-ExpressionType* *s* :

- *Indexed-ExpressionType*
  - *Register-ExpressionType*
- 

## Definition

*Initializer-ExpressionType* :

*Scalar-Initializer-ExpressionType*  
*Compound-ExpressionType*  
*Duplicated-ExpressionType*

*Scalar-Initializer-Expression Type :*  
*Absolute-Expression Type*  
*Constant-Expression Type*  
*Immediate-Expression Type*  
*Direct-Expression Type*  
*Indirect-Expression Type*  
*String-Expression Type*  
*Floating-Point-Expression Type*  
*Type-Expression Type*

---

## Text Preprocessor

The text preprocessor is a functional unit within the assembler that performs the *text preprocessing* translation phase. During text preprocessing, the following actions are performed:

1. [Language Elements](#) are recognized.
2. Text equates and macros are expanded.
3. [Macro directives](#) and [conditional assembly directives](#) are recognized and processed.
4. The preprocessed output is passed on to the assembler for final processing.

This section also describes the various types of preprocessor directives:

Type	Function	Directives
Conditional Assembly	Tests for a specified condition and assembles a block of statements if the condition is true.	IF IFB IFDEF IFDIFI IFE IFIDN IFNB IFNDEF IF1 IF2 ELSE ENDIF
Text Equate	Allows assignment of simple text strings to a symbolic name. Provides functions for expanding and operating on the values.	CATSTR EQU INSTR SIZESTR SUBSTR
Macro	Provides text processing that is done sequentially at assembly time. By the end of assembly, ALP expands all macros and assembles the resulting text into object code.	ENDM EXITM FOR FORC IRP IRPC LOCAL MACRO PURGE REPEAT REPT
Miscellaneous	Miscellaneous text processing functions.	COMMENT ECHO %OUT INCLUDE

---

## Text Operators

---

## Description

The [Text Preprocessor](#) recognizes certain punctuation characters as text operators. The programmer may use these operators to force the [Text Preprocessor](#) to perform various operations such as delineating text, expanding arguments, and converting expressions into their text representations.

---

## Syntax

*Text-Operator :*  
*Literal-Character-Operator*  
*Literal-Text-Operator*  
*Text-Expansion-Operator*  
*Text-Substitution-Operator*

---

## Literal Character Operator (!)

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

*Literal-Character-Operator :*  
! any printable character

---

## Description

When you use an exclamation point (!) in an operand, ALP treats the next character literally. (!) is typically used to prevent the assembler from recognizing and acting upon special characters such as the semicolon (;) or the ampersand (&), forcing them to appear as normal data

characters.

---

## Constraints

The *Literal-Character-Operator* has no effect when used inside of a *String-Literal*.

---

## Examples

In this example, use of the ! in the second macro argument prevents the assembler from interpreting the rest of the line as a comment:

```
MACRONAME First, !;NonComment, Third          ;Comment
```

---

## Literal Text Operator ()

---

## Related Information

[Description](#)

[Examples](#)

---

## Syntax

*Literal-Text-Operator* :  
    < *Char-Sequence* >

*Char-Sequence*  
    any printable character  
    *Char-Sequence* any printable character

---

## Description

The literal-text operator directs the assembler to treat *Char-Sequence* as a single literal element regardless of whether it contains commas, spaces, or other separators. The operator is most often used with macro calls and the FOR directive to ensure that values in a parameter list are treated as a single parameter.

The literal-text operator can also be used to force ALP to treat other special characters such as the semicolon (;) or the ampersand (&) literally. For example, the semicolon inside angle brackets (<;>) becomes a semicolon, not a comment indicator.

ALP removes one set of angle brackets each time the parameter is used in a macro. When using nested macros, you will need to supply as many sets of angle brackets as there are levels of nesting. The assembler recognizes nested occurrences of text literals.

---

## Examples

The following example illustrates how to pass arbitrary text to a macro as a single parameter:

```
MACRONAME First, <Second Argument>, <Third, <Nested>, Argument>
```

The macro will receive three separate arguments:

1. First
2. Second Argument
3. Third, <Nested>, Argument

Notice that the outermost set of angle brackets were removed from the second and third arguments.

---

## Text Expansion Operator (%)

---

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

## Syntax

```
Text-Expansion-Operator :  
    % 2nd through Nth token on line  
    % Text-EquateName  
    % Expression
```

---

## Description

The % *Text-Expansion-Operator* has different effects depending upon the context in which it is used. Its primary purpose is convert various sources of information into text literals that may in turn be passed to macros as arguments.

The % *Text-Expansion-Operator* causes the following types of conversions:

### Line Expansion

When used as the first token on the line, the % operator forces expansion of *Text-EquateName* s in contexts where they would otherwise be left unexpanded. *Text-EquateName* s passed as arguments to macros are not automatically expanded; this is one context where the % operator is useful.

### Expansion of a Text Equate Operand

As with *Line Expansion*, the % operator may be used within the body of a line to expand individual *Text-EquateName* s. This can be useful when expansion of all *Text-EquateName* s on the line is not desired.

### Conversion of Numeric Expression to Text

If the *Text-Expansion-Operator* is not the first token on the line or immediately followed by a *Text-EquateName*, then the argument of the % operator is assumed to be an *Expression*, which is evaluated and converted to the text representation of its value. This is useful when the need arises to pass the text representation of a number to a macro.

-----

## Constraints

When the % *Expression* form of the expansion operator is used, the *Expression* must evaluate to an *Immediate-ExpressionType*.

-----

## Examples

```
MakErr      MACRO      X
LB           =          0
             REPEAT    X
LB           =          LB+1
             MakLib    %LB
             ENDM      ;;End of REPEAT
             ENDM      ;;End of MACRO
MakLib      MACRO      Y
Err&Y: DB    'Error  &Y',0
             ENDM

             MakErr 3
Err1: DB    'Error 1',0
Err2: DB    'Error 2',0
Err3: DB    'Error 3',0
```

-----

## Text Substitution Operator (&)

-----

## Related Information

[Description](#)

[Constraints](#)

[Examples](#)

---

# Syntax

*Text-Substitution-Operator* :  
*Macro-ParameterName* &  
&*Macro-ParameterName*

---

## Description

An ampersand (&) is used in the body of a macro to force the substitution of a *Macro-ParameterName* with the value of its argument during expansion of the macro.

---

## Constraints

The assembler does not substitute a *Macro-ParameterName* that is in a quoted string or not preceded by a delimiter in the expansion unless it is immediately preceded by an ampersand (&).

It is necessary to separate a *Macro-ParameterName* from other *Identifier-Character*s with an ampersand (&) before any substitution or paste operations are performed.

---

## Examples

```
ErrGen      MACRO      X
Error&X:    push      bx
ABX         mov        BX, "A"
AB&X       jmp        ERROR
            ENDM
```

The statement **ErrGen A** produces this code:

```
ErrorA:    push      bx
ABX        mov        BX, "A"
ABA        jmp        ERROR
```

---

## Preprocessor Tokens

---

## Description

During the text preprocessing translation phase, certain conditions will cause the preprocessor to convert raw [Language Elements](#) (*Token*s) into *Preprocessing-Token*s. The act of text preprocessing typically causes *Preprocessing-Token*s to either be removed from the input stream or converted back into *Token*s before being passed on to the assembler for final processing.

---

## Syntax

*Preprocessing-Token* :  
[Identifier](#)  
[Text-Literal](#)  
[FileName](#)  
[Comment](#)

---

## Text Literals

## Related Information

[Description](#)

[Constraints](#)

---

## Syntax

*Text-Literal* :  
operand of [Literal-Character-Operator](#)  
operand of [Literal-Text-Operator](#)

---

## Description

A *Text-Literal* is a single unit of text that is used by the [Text Preprocessor](#) in many different text handling contexts. In some contexts (such as the processing of arguments to be passed to a macro), normal language *Token*s are implicitly treated as *Text-Literal*s, provided they are not a delimiter character such as a comma or a blank. In other contexts, it may be necessary to explicitly convert a unit of text to a *Text-Literal* using the [Literal-Text-Operator](#).

---

## Constraints

A normal language *Token* is never implicitly considered to be a *Text-Literal* if a *Text-Literal* is explicitly required in the syntax of the construct being parsed.

---

## File Names

---

## Related Information

[Description](#)

[Examples](#)

---

## Syntax

```
FileName .:  
    FileName-Text  
    Text-Literal  
  
FileName-Text .:  
    FileName-Character  
    FileName-Text FileName-Character  
  
FileName-Character .:  
    any printable character except blank (ASCII 32)
```

---

## Description

*FileName* arguments may be coded as an arbitrary sequence of printable characters, or as a *Text-Literal*; use the *Text-Literal* form if the *FileName* is to contain embedded spaces or other special characters.

If path information is included in the *FileName*, you can separate the individual directory names with either the back slash (\) or the forward slash (/) and they will be treated identically by the assembler.

---

## Examples

```
INCLUDE    <inc\macros.inc>  
INCLUDELIB os2386.lib
```

---

# Comments

---

## Related Information

[Description](#)

[Examples](#)

---

## Syntax

```
Comment :  
    EndOfLine-Comment  
    Block-Comment  
  
EndOfLine-Comment :  
    NonMacro-Comment  
    Macro-Comment  
  
NonMacro-Comment :  
    ; Char-Sequence  
  
Macro-Comment :  
    ;; Char-Sequence  
  
Char-Sequence :  
    any printable character  
    Char-Sequence any printable character  
  
Block-Comment :  
    See the COMMENT directive
```

---

## Description

**Comments** are language elements that have significance only to the programmer and not to the assembler. Comments are effectively removed from the input stream during the text preprocessing phase.

There are two classes of comments recognized by ALP:

- Comments that start with a character sequence and continue to the end of the line (*EndOfLine-Comment*)
- Comments that start with a character sequence and continue until the occurrence of another character sequence (*Block-Comment*). See the [COMMENT](#) directive for a description of *Block-Comments*.

There are two types of *EndOfLine-Comments*:

### **Macro-Comment**

**Macro-Comments** (beginning with two semicolons) do not appear in the listing output even when the .LALL directive is used. Use of *Macro-Comments* can significantly reduce the amount of memory workspace used by the definition of a macro. As a macro definition is read, *Macro-Comments* are discarded and not entered into the macro definition,

whereas *NonMacro-Comments* are treated as normal text and are retained.

### *NonMacro-Comment*

*NonMacro-Comment* (beginning with a single semicolon) are preserved in macro definitions and appear in the listing output during macro expansions.

---

## Examples

The following are examples of *EndOfLine-Comments*:

```
; Comments may be on a line all by themselves. They can be empty...
;
BumpCount MACRO Amount          ; They don't have to start in the first column
    Count = Count + Amount      ; They can appear to the right of statements
    $Total = $Total + Amount    ; This appears in macro expansions
ENDM                             ;;This does not, discarded during definition
```

---

## Text Arguments

---

## Description

Many preprocessing directives operate on sequences of raw text characters called *Text-Arguments*. A *Text-Argument* may be specified using any one of several methods:

- Specifying the text directly using a raw *Text-Literal*.
- Using the *Text-Expansion-Operator* to convert a numeric expression to its text representation.
- Using a *Text-EquateName* in those contexts where a *Text-Argument* is expected. In this case the preprocessor will automatically resolve the *Text-EquateName* and use its value as the *Text-Argument*.

---

## Syntax

*Text-Argument* :  
    *Text-Literal*  
    % *Expression*  
    *Text-EquateName*

---

## Conditional Assembly Directives

At assembly time, ALP evaluates conditional assembly directives, assembling if the conditions are true. You can use conditional assembly directives when you want to test for a specified condition and assemble a block of statements if the condition is true. The **IFxx** and **ENDIF** directives enclose the statements to be considered for conditional assembly. The optional **ELSEIFxx** and **ELSE** blocks follow the **IFxx** directive. There are many forms of the **IFxx** and **ELSEIFxx** directives.

This section describes the following conditional assembly directives:

**IF**  
**IFB**  
**IFDEF**  
**IFDIF**  
**IFDIFI**  
**IFE**  
**IFIDN**  
**IFIDNI**  
**IFNB**  
**IFNDEF**  
**IF1**  
**IF2**  
**ELSE**  
**ENDIF**

---

## IFxx (Begin Primary Conditional Block)

You can use each **IFxx** conditional directive with the **ELSExx**, **ELSE** and **ENDIF** directives to provide the statements to be considered for conditional assembly. ALP assembles the statements following the **IFxx** directive only if this condition is true.

### Syntax

```
IFxx operand
.
.
.
[ELSEIFxx] (optional)
.
.
.
[ELSE] (optional)
.
.
.
ENDIF
```

### Remarks

The following directives are members of the **IFxx** family:

- **IF**
- **IFB**
- **IFDEF**
- **IFDIF**
- **IFDIFI**
- **IFE**
- **IFIDN**
- **IFIDNI**
- **IFNB**
- **IFNDEF**
- **IF1**
- **IF2**

You can nest the conditional directives to any level. They are not limited to use within a macro. The assembler must know any operand to a conditional on pass one to avoid errors and incorrect evaluation.

---

# IF (If Expression is True)

**IF** starts a conditional assembly statement, which is ended by the corresponding **ENDIF** conditional assembly directive. Each **IF** directive must be ended by a matching **ENDIF** directive.

## Syntax

```
IF Expression
.
.
.
[ELSEIFxx] (optional)
.
.
.
[ELSE] (optional)
.
.
.
ENDIF
```

## Remarks

If the **IFxx** conditional assembly statement is not ended by an **ENDIF** directive, an *unterminated conditional* message is produced by the assembler. An **ENDIF** without a matching **IF** causes an error. **ENDIF** does not have an operand.

**Note:** The conditional directives can be nested to any level. They are not limited to use within a macro. Any operand to a conditional must be known on pass 1 to avoid errors and incorrect evaluation.

## Example

```
IF debug
    EXTERN dump:FAR
    EXTERN trace:FAR
    EXTERN breakpoint:FAR
ENDIF
```

-----

# IFB (If Argument is Blank)

This is true if *Text-Argument* is blank (contains no characters).

## Syntax

```
IFB Text-Argument
```

## Remarks

A *Text-Argument* must be specified, the contents of which are checked for the presence of characters. An error is generated if a *Text-Argument* is not supplied.

-----

# IFDEF (If Identifier is Defined)

This is true if *Identifier* has been defined as a label, variable, or symbol.

## Syntax

```
IFDEF Identifier
```

---

# IFDIF (If Arguments Are Different)

This is true if *Text-Argument -1* and *Text-Argument -2* are different in a case-sensitive comparison.

## Syntax

```
IFDIF Text-Argument -1, Text-Argument -2
```

## Remarks

Both *Text-Argument* arguments must be specified. An error is generated if a either argument is not supplied.

## Example

In the following example:

```
IFDIF <EAGLES>,<Eagles>  
    value = 1  
ENDIF
```

the condition would be true; the arguments are different because they are compared with a case-sensitive algorithm.

---

# IFDIFI (If Arguments Are Spelled Differently)

This is true if *Text-Argument -1* and *Text-Argument -2* are different in a case-insensitive comparison.

## Syntax

```
IFDIFI Text-Argument -1, Text-Argument -2
```

## Remarks

Both *Text-Argument* arguments must be specified. An error is generated if a either argument is not supplied.

## Example

In the following example:

```
IFDIFI <EAGLES>,<Eagles>
    value = 1
ENDIF
```

the condition would be false; the arguments are not different because they are compared using a case-insensitive algorithm.

---

## IFE (If Expression is Not True)

This is true if *expression* is 0.

### Syntax

```
IFE Expression
```

---

## IFIDN (If Arguments Are Identical)

This is true if *Text-Argument -1* and *Text-Argument -2* are identical in a case-sensitive comparison.

### Syntax

```
IFIDN Text-Argument -1, Text-Argument -2
```

### Remarks

Both *Text-Argument* arguments must be specified. An error is generated if a either argument is not supplied.

### Example

In the following example:

```
IFIDN <EAGLES>,<Eagles>
    value = 1
ENDIF
```

the condition would be false; the arguments are not identical because they are compared using a case-insensitive algorithm.

---

## IFIDNI (If Arguments Are Spelled Identically)

This is true if *Text-Argument -1* and *Text-Argument -2* are identical in a case-insensitive comparison.

### Syntax

IFIDNI *Text-Argument-1*, *Text-Argument-2*

#### Remarks

Both *Text-Argument* arguments must be specified. An error is generated if a either argument is not supplied.

#### Example

In the following example:

```
IFIDNI <EAGLES>,<Eagles>
    value = 1
ENDIF
```

the condition would be true; the arguments are identical because they are compared using a case-insensitive algorithm.

---

## IFNB (If Argument is Not Blank)

This is true if *Text-Argument* is not blank (characters are present).

#### Syntax

IFNB *Text-Argument*

#### Remarks

A *Text-Argument* must be specified, the contents of which are checked for the presence of characters. An error is generated if a *Text-Argument* is not supplied.

---

## IFNDEF (If Identifier is Not Defined)

This is true if *symbol* has not yet been defined as a label, variable, or symbol.

#### Syntax

IFNDEF *symbol*

---

## IF1 (If Assembling On Pass 1)

This is true on pass one.

#### Syntax

IF1

#### Remarks

IF1 does not have an operand.

-----

## IF2 (If Assembling On Pass 2)

This is true on pass two.

#### Syntax

IF2

#### Remarks

IF2 does not have an operand.

-----

## ELSEIFxx/ELSE (Begin Alternate Conditional Block)

Each conditional directive can be used with the **ELSE** directive to provide the statements to be considered for conditional assembly. The **ELSE** directive allows the assembly of the statements following it when the IFxx condition or intervening **ELSEIFxx** conditions are false.

#### Syntax

```
IFxx
.
.
.
[ELSEIFxx] (optional)
.
.
.
[ELSE] (optional)
.
.
.
ENDIF
```

#### Remarks

There is a corresponding **ELSEIFxx** directive to match all forms of the IFxx family of directives:

- ELSEIF
- ELSEIFB
- ELSEIFDEF
- ELSEIFDIF
- ELSEIFDIFI
- ELSEIFE
- ELSEIFIDN
- ELSEIFIDNI
- ELSEIFNB

- ELSEIFNDEF
- ELSEIF1
- ELSEIF2

For information about the meaning of the conditional tests performed by the **ELSEIFxx** directives, refer to the definitions for the corresponding **IFxx** directives.

Any number of **ELSEIFxx** blocks may be used within a given **IFxx** statement. Only one **ELSE** block is permitted for a given **IFxx**. A conditional directive with more than one **ELSE** or an **ELSE** without a conditional directive causes an error. **ELSE** does not have an operand.

**Note:** The conditional directives can be nested to any level. They are not limited to use within a macro. Any operand to a conditional must be known on pass 1 to avoid errors and incorrect evaluation.

### Example

```
IF DEFBUF
    BUF DB 100 DUP(0)
ELSE
    EXTERN BUF:BYTE
ENDIF
```

## ENDIF (End a Conditional Assembly Statement)

**ENDIF** ends the conditional assembly statement begun by the corresponding **IFxx** conditional assembly directive. Each **IFxx** directive must be ended by a matching **ENDIF** directive.

### Syntax

```
IFxx
.
.
.
[ELSEIFxx] (optional)
.
.
.
[ELSE] (optional)
.
.
.
ENDIF
```

### Remarks

If the **IFxx** conditional assembly statement is not ended by an **ENDIF** directive, an **unterminated conditional** message is produced by the assembler. An **ENDIF** without a matching **IFxx** causes an error. **ENDIF** does not have an operand.

**Note:** The conditional directives can be nested to any level. They are not limited to use within a macro. Any operand to a conditional must be known on pass 1 to avoid errors and incorrect evaluation.

### Example

```
IF debug
    EXTERN dump:FAR
    EXTERN trace:FAR
    EXTERN breakpoint:FAR
ENDIF
```

---

## Text Equate Directives

A *Text Equate* is a symbolic name you give to a series of characters. Text equates are used to expand text within a source statement. The directives described in this section create and manipulate text equates.

```
EQU
CATSTR
INSTR
SIZESTR
SUBSTR
```

---

## CATSTR (Concatenate Strings)

**CATSTR** concatenates a list of text values specified by *string* into a single text value and assigns it to *Name*.

### Syntax

```
Name CATSTR string [, string] ...
```

---

## EQU Directive (Assign Text to a Symbolic Constant)

The **EQU** directive assigns the contents of a text literal to *Name*.

### Syntax

```
Name EQU Text-Literal
```

### Remarks

The value of the *Text-Literal* is assigned to the *Name* entry. In normal contexts, subsequent references to *Name* will cause the preprocessor to replace *Name* with the value specified by the *Text-Literal* entry. This is a simple text substitution operation.

The *Name* entry is a globally-scoped *Identifier* that is converted to a *Text-EquateName*. The *Name* cannot have been previously defined as a different *Identifier-Type*. However, the *Name* entry can be redefined as many times as desired with different values for the *Text-Literal* entry.

See also **EQU** and **=**.

### Example

```
A    EQU    <BP +>    ;explicit text literal, A is a text equate
A    EQU    <3>        ;redefinition of A with different value
```

---

## INSTR (Search In String For Value)

**INSTR** searches a specified *String* for an occurrence of a given *Sub-String* and assigns its position (1-based) to *Name*. The search is case sensitive. *Start* is the position in *String* to start the search for *Sub-String*. If *Start* is not given, it is assumed to be 1 (the start of the string). If *Sub-String* is not found, the position assigned to *Name* is 0.

### Syntax

```
Name INSTR [Start,]String,Sub-String
```

### Remarks

**INSTR** assigns the position value to a name as if it were a numeric equate.

### Example

```
pos INSTR <person>, <son>
```

---

## SIZESTR (Return Size Of String)

Assigns the number of characters given by the *Text-Argument* to *Name*.

### Syntax

```
Name SIZESTR Text-Argument
```

---

## SUBSTR (Extract a Sub-string From a String)

Assigns a substring of *Text-Argument* starting at *Position* to the symbol given by *Name*.

### Syntax

```
Name SUBSTR Text-Argument,Position[,Length]
```

## Remarks

The *Position* parameter indicates the starting character of the substring to extract from the *Text-Argument*, and must be 1 or greater. If specified, the *Length* parameter indicates how many characters are desired, otherwise the remainder of the string is extracted.

---

# Macro Directives

A **macro procedure or function**, which is comprised of one or more statements.

Macro processing is text processing that is done sequentially at assembly time. By the end of assembly, ALP expands all macros and assembles the resulting text into object code.

This section describes the following types of macros:

- Macro procedures, which expand to one or more complete statements and can optionally take parameters
- Repeat blocks, which generate a group of statements a specified number of times or until a condition becomes true

This section describes the following macro directives:

ENDM  
EXITM  
FOR/IRP  
FORC/IRPC  
LOCAL  
MACRO  
PURGE  
REPEAT/REPT

---

## ENDM (End Current Macro Definition)

End each **MACRO**, **REPEAT/REPT**, **FOR/IRP**, and **FORC/IRPC** directive with the **ENDM** directive.

## Syntax

**ENDM**

## Remarks

If the **ENDM** directive is not used with the **MACRO**, **REPEAT/REPT**, **FOR/IRP**, and **FORC/IRPC** directives, an error occurs. An unmatched **ENDM** also causes an error.

If the assembler produces an error message stating that it found the end-of-file on the source and cannot find an **END** statement when there was an **END**, the likely cause is a missing **ENDM** or **ENDIF** statement. Without **ENDM**, the assembler treats the rest of the source as part of the **MACRO** definition.

**Note:** The *name field* is not allowed. Do not confuse the **ENDM** directive with other ending directives that do require the name of the block being ended, such as **ENDP** or **ENDS**.

## Example

```
addup  MACRO    ad1,ad2,ad3
        MOV     AX,ad1      ;;first parameter in AX
        ADD     AX,ad2      ;;add next two parameters
        ADD     AX,ad3      ;;leave sum in AX
```

## EXITM (End Current Macro Expansion)

Use the **EXITM** directive when a block contains a directive that tests for some condition and you want to end the current macro expansion when the test proves that the remainder of the expansion is not required. When an **EXITM** directive is run, the expansion is stopped immediately, and any remaining expansion or repetition is not produced.

### Syntax

```
EXITM
```

### Remarks

Only the block containing the **EXITM** directive is ended; outer levels of a nested macro expansion continue unaffected.

**EXITM** is executed at macro expansion time and is not a substitute for the **ENDM** directive, which marks the end of the macro body and is recognized at macro definition time.

### Example

```
DSEG SEGMENT
    .
    .
    .
SYM = 0
    REPEAT 16
;;Check for paragraph boundary
    IF ($-DSEG) MOD 16 EQ 0
        EXITM ;;quit if padded to boundary
    ENDIF
SYM = SYM + 1
    DB SYM ;;produce numbered padding
ENDM
```

## FOR/IRP (Iterative Macro Expansion Using List of Arguments)

The **FOR** directive, used in combination with the **ENDM** directive, designates a block of statements to be repeated, once for each argument in the list enclosed by angle brackets. Each repetition substitutes the next item in the *<Argument-List>* entry for every occurrence of *Parameter* in the block.

### Syntax

```
FOR Parameter, <Argument-List>
    .
    .
    .
ENDM
```

### Remarks

The obsolete spelling for the **FOR** directive is **IRP**.

You must enclose the *<Argument-List>* entry in angle brackets. It has the following format:

```
<[Argument [, Argument ...]]>
```

If an empty (*<>*) *Argument* is found in *<Argument-List>*, the *Parameter* name is replaced by a null value. If the argument list is empty, the **FOR** directive is ignored and no statements are copied. The assembler processes the block once for each *Argument* in the *<Argument-List>*, replacing each occurrence of *Parameter* in the macro body with the current *Argument* value.

The **FOR/IRP-ENDM** block does not have to be within a macro definition.

### Example

In this example, the assembler produces the code **DB 1** through **DB 10**.

```
FOR    X, <1,2,3,4,5,6,7,8,9,10>
DB     X
ENDM
```

In the next example:

```
FOR    ARGUMENT,<"first line",13,10,
"second line",13,10>
DB     ARGUMENT
ENDM
```

The assembler produces the code:

```
DB     "first line"
DB     13
DB     10
DB     "second line"
DB     13
DB     10
```

-----

## FORC/IRPC (Iterative Macro Expansion Using List of Characters)

The assembler repeats the statements in the block once for each character in the string. Each repetition substitutes the next character in the string for every occurrence of *Parameter* in the block.

### Syntax

```
FORC Parameter, String (or <String>)
.
.
.
ENDM
```

### Remarks

The obsolete spelling for the **FORC** directive is **IRPC**.

The **FORC** directive is similar to the **FOR/IRP** directive except that a *String* is used instead of *<Argument-List>*, and the angle brackets around the string are optional. The string should be enclosed with angle brackets (*<>*) if it contains spaces, commas, or other separating characters.

The **FORC/IRPC-ENDM** block does not have to be within a macro definition.

### Example

In this example, the assembler produces the code **DB 1** through **DB 8**:

```
FORC      X,12345678
DB         X
ENDM
```

---

## LOCAL (Identify Names Local to a Macro Definition)

The **LOCAL** directive is used inside the body of a macro definition, and provides a method of automatically generating unique assembler labels each time the macro is expanded. The names appearing in the argument list of the **LOCAL** directive are known only to the enclosing macro, and each time they are referenced during a macro expansion a unique symbol is created. This prevents the assembler from issuing duplicate definition errors when the macro is expanded more than once and symbols contained therein are being used to create assembler labels.

### Syntax

```
LOCAL Name [, Name....]
```

### Remarks

The **LOCAL** directive is recognized only within the body of a macro given by a **MACRO**, **FOR/IRP**, **FORC/IRPC**, or **REPEAT/REPT** definition. The symbols created by the preprocessor are of the form **??nnnn**, where **nnnn** is a hexadecimal number in the range 0000 through FFFF. You must avoid using identifiers of this form for your own purposes, because doing so can cause duplicate definition errors.

To insure that they have the proper effect, **LOCAL** statements should appear in the body of the macro before any other directives are used. It is acceptable for blank lines or comments to precede any **LOCAL** statements.

You can use multiple **LOCAL** statements if the argument list is too long to fit on one line, or if you want a vertical list of **LOCAL** symbols.

### Example

```
DISPLAY MACRO TT

; Blank lines and comments are ok here
    LOCAL AGAIN

;; DOS macro to display message addressed by BX TT times
    MOV     CX,TT
    MOV     AH,9
    MOV     DX,BX
;Generate a unique label for AGAIN
AGAIN:
    INT     21H
    LOOP    AGAIN
ENDM
```

---

## MACRO (Assign a Body of Text to a Name)

This directive produces a given sequence of statements from various places in your program, even though different parameters may be required each time you call the sequence.

Macro processing consists of two separate and distinct phases: [Macro Definition](#) and [Macro Expansion](#).

---

## Macro Definition

A macro definition consists of three essential parts:

- The **MACRO** directive, defining the *Name* and the *Parameter-List*
- The body of the macro, containing the prototypes of statements to produce when you invoke the macro for expansion.
- The **ENDM** directive, ending the definition of the macro.

### Syntax

```
Name MACRO [Parameter [, Parameter ...]]  
.  
.  
.  
ENDM
```

### Remarks

The *Name* field must be a valid preprocessor identifier and specifies the symbolic name that the user will refer to when invoking the macro for expansion. If *Name* is already defined, it must be that of a previous macro definition, otherwise an error message is issued. Macros may be redefined to have a different *Parameter-Lists* or macro body text; doing so causes the previous definition to be lost.

The optional *Parameter-List* is the complete comma-separated list of all *Parameter* values given in the macro definition statement. A parameter must be a valid symbol name according to the rules for naming preprocessor and assembler identifiers. Each parameter becomes a symbol that is local to the macro being defined and is recognized during macro expansion prior to searching the global name space. Thus, macro parameters need not have names unique from identifiers defined elsewhere in the program.

---

## Macro Expansion

To expand the macro, the macro *Name* (defined in the *Name* field of the **MACRO** definition statement) is coded as you would any other assembler directive, followed by the list of arguments (if any) that you want to pass to the macro.

### Syntax

```
Name [Argument [, Argument ...]]
```

### Remarks

The *Name* field must be the name of a macro defined previously with a **MACRO** directive.

Each *Argument* field denotes a text value that you want to pass to the macro. The relative positions of the elements are important, because each *Argument* is associated in left-to-right fashion with the corresponding *Parameter* as defined in the *Parameter-List* during the macro definition.

The number of *Argument* entries given when the macro is invoked need not be the same as the number of *Parameter* entries. If you pass extra *Arguments* to the macro, they are ignored; if too few are supplied, empty text values are associated with the remaining *Parameters*. You may also associate an empty text value with a *Parameter* by passing an explicitly empty text literal <> as an *Argument*.

Commas are normally used to separate arguments, although blanks or tabs are also considered to be argument separators. For this reason, any argument that must contain an argument separator character (commas, blanks, or tabs) should be enclosed in angle brackets <>. For example:

```
PUSHVEC    MACRO  PARM1, PARM2
            MOV    AX, PARM1
            PUSH   AX
            MOV    AX, PARM2
            PUSH   AX
            ENDM
            .
            .
            .
            PUSHVEC DS, <OFFSET VARNAME>
; PUSH DWORD VECTOR OF VARNAME ONTO STACK
```

You can also use angle brackets to produce variable lengths of results. For example:

```
STRING     MACRO   NUMBERS
            DB      NUMBERS
            ENDM
            .
            .
            .
            STRING <1, 2, 3, 4>
; PRODUCE 4 BYTES OF INTEGER NUMBERS
```

## Remarks

Each time a macro is invoked (expanded) by specifying its name, the preprocessor emits the statements contained in the body of the macro and passes them to the assembler for processing. During the expansion process, any replacement parameters encountered in the macro body (as named in the *Parameter-List* of the macro definition) are replaced with the corresponding *Argument* (if any) passed through the *argument-list* at the time the macro was invoked.

## Example

```
GEN        MACRO   XX, YY, ZZ
            MOV     AX, XX
            ADD     AX, YY
            MOV     ZZ, AX
            ENDM
```

When the call is made, for example:

```
GEN    ED, KISER, SUM
```

The assembler produces the following code:

```
MOV     AX, ED
ADD     AX, KISER
MOV     SUM, AX
```

---

# PURGE (Remove Macro Definition)

The **PURGE** directive deletes the definition of a specified macro entry, letting you reuse space.

## Syntax

```
PURGE Macro-Name [, ...]
```

## Remarks

It is not necessary to purge a macro before redefining it. You may use **PURGE** to recover memory during assembly by deleting the contents of unreferenced macros. An **Out of Memory** condition can occur if a large, general-purpose macro library is included.

## Example

The directive:

```
PURGE    MACRONAME
```

performs the same function as redefining the macro with no contents, as in:

```
MACRONAME MACRO
          ENDM
```

In the following example, assume that MAC1 is a macro included in MACRO.LIB:

```
INCLUDE  MACRO.LIB
PURGE    MAC1
MAC1     ; Calls the purged macro
          ; but produces nothing
```

-----

# REPEAT/REPT (Iterative Macro Expansion Using a Count Expression)

**REPEAT** specifies the number of times to generate the statements inside the macro.

## Syntax

```
REPEAT Expression
      Statements
ENDM
```

## Remarks

The *Expression* field must evaluate to an *Absolute-ExpressionType* (it cannot contain forward references). Because the repeat block will be expanded at assembler time, the number of iterations must be known then.

-----

# ECHO Directive (Display Message on Standard Output Device)

The **ECHO** directive displays progress through a long assembly or displays the value of conditional assembly parameters.

## Syntax

**ECHO** *Text*

## Remarks

The assembler lists the *Text* entry on the standard output device during assembly when the assembler encounters the **ECHO** directive.

**ECHO** is not available under MASM 5.10 emulation; you must use **%OUT**, which is the obsolete spelling for the **ECHO** directive.

## Example

### Example 1:

```
IF IBM
    ECHO IBM VERSION
ENDIF

IF2
    ECHO STARTING SECOND PASS
    .
    .
    .
ENDIF
```

### Example 2:

```
INNER    MACRO    TEXT, VAL
          ECHO     TEXT VAL
          ENDM
          .
          .
          .
HERE     =         $ - CSEG
          INNER    <CURRENT LOCATION>, %HERE
```

---

# INCLUDE Directive (Insert File Contents into Input Stream)

The **INCLUDE** directive "stacks" the current source file and begins reading tokens from the source file given by the *FileName* argument. If you use the **INCLUDE** directive, you need not repeat a sequence of statements that are common to several source files.

## Syntax

**INCLUDE** *FileName*

## Remarks

The assembler uses the following search order when attempting to open the **INCLUDE** file:

1. If the *FileName* argument contains a fully qualified path name (one that begins with a back slash or forward slash), then the assembler attempts to open the file exactly as specified, and no other search is performed if the file is not found.
2. If the *FileName* begins with a relative path name or contains no path information, the assembler begins searching for the **INCLUDE** file by looking in the directory of the source file that issued the **INCLUDE** directive.
3. The assembler searches for *FileName* in the list of directories given by any **-Fdi** or **-I** options found on the command line.

4. The assembler searches for *FileName* in the list of directories given by the <BaseEXE>\_INCLUDE environment variable.
5. The assembler searches for *FileName* in the list of directories given by the INCLUDE environment variable.
6. Lastly, the assembler searches for *FileName* in the current directory. If the named file is not found, the assembler issues a fatal error message and the assembler is ended.

In no case does the assembler strip relative path information from the *FileName* when performing search steps 2 through 6.

When the file named in the **INCLUDE** directive is located, the assembler opens it and assembles all of the statements contained therein until the end of the file is reached. The file is then closed and assembler resumes in the original module at the line following the **INCLUDE** directive.

An **INCLUDE** file should not contain an **END** assembler directive to denote the end of the included module; the assembler closes the included module when its physical end of file is reached.

**INCLUDE** files may be nested to any reasonable level, and is limited only by the operating system's ability to provide the necessary resources.

#### Example

```
INCLUDE OS2.INC
```

-----

## COMMENT Directive (Program Information Block)

**COMMENT** lets you enter comments about your program without having to enter semicolons (;) for each line.

#### Syntax

```
COMMENT Delimiter Text Delimiter
```

#### Remarks

The first non-blank character after **COMMENT** is the first delimiter. The **COMMENT** directive causes the assembler to treat all *Text* between *Delimiter* and *Delimiter* as a comment. The text must not contain the delimiter character. This directive is used for multiple-line comments. A **COMMENT** defined in the body of a macro does not appear unless **.LALL** is requested.

#### Example

```
COMMENT *You can enter as many lines  
of text between the delimiters  
.  
.  
.  
as you need to describe your program.*
```

-----

## Assembler Directives

This section describes the various types of ALP directives:

Type	Function	Directives
Conditional error	Debugs programs and checks for assembly-time errors.	.ERR .ERR1 .ERR2 .ERRDEF .ERRNDEF .ERRE .ERRNZ .ERRB .ERRDIF .ERRDIFI .ERRIDN .ERRIDNI .ERRNB
Data allocation	Allows you to create and initialize variables for use within your program.	BYTE DB DD DF DQ DT DW DWORD FWORD QWORD REAL4 REAL8 REAL10 SBYTE SDWORD SWORD TBYTE WORD
Intermodule linkage	Simplifies data sharing and provides a high-level interface to multiple-module programming.	COMM END EXTERN/EXTRN EXTERNDEF INCLUDELIB NAME PUBLIC
Listing control	Controls the assembler listing of your source file.	%BIN .CREF .LALL .LIST .LISTALL .LISTIF .LISTMACRO .LISTMACROALL .NOCREF .NOLIST .NOLISTIF .NOLISTMACRO PAGE .SALL .SFCOND SUBTITLE SUBTTL .TFCOND TITLE .XALL .XCREF .XLIST
Procedure control	Allows you to organize your code into procedures.	PROC LOCAL ENDP
Processor control	Selects processors and coprocessors.	.186 .286 .286P .287 .386 .386P .387 .486

		.486P .586 .586P .686 .686P .8086 .8087 .MMX .NOMMX
Segments	Creates and manages segments.	ALIGN .ALPHA .CODE .CONST .DATA .DATA? DOSSEG .DOSSEG ENDS EVEN .FARDATA .FARDATA? GROUP .MODEL ORG SEGMENT .SEQ .STACK
Type definition	Allows the creation of complex user-defined data types.	RECORD STRUC STRUCT TYPEDEF UNION
Miscellaneous	Provides miscellaneous functions.	= .ABORT ASSUME EQU LABEL OPTION .RADIX

-----

## Conditional Error Control

Use conditional error control directives to debug programs and check for assembly-time errors. If you insert a conditional assembly directive in your code, you can test assembly-time conditions at that point. You can also test for boundary conditions in macros by using conditional error control directives.

Errors generated by conditional error control directives cause ALP to return a nonzero return code. If a severe error is detected during assembly, ALP does not generate the object module.

This section describes the following conditional error control directives:

.ERR  
.ERR1  
.ERR2  
.ERRB  
.ERRDEF  
.ERRDIF  
.ERRDIFI  
.ERRE  
.ERRIDN  
.ERRIDNI  
.ERRNB  
.ERRNDEF  
.ERRNZ

-----

# .ERR/.ERR1/.ERR2 (Force Assembly Error Condition)

The **.ERR**, **.ERR1**, and **.ERR2** directives cause errors at the points at which they occur in the source file.

## Syntax

```
.ERR  
    OR  
.ERR1  
    OR  
.ERR2
```

## Remarks

The **.ERR** directive causes an error regardless of the pass. **.ERR1** causes an error on the first pass only. **.ERR2** causes an error on the second pass only. If you use the **-Lp:1** option to request a first pass listing, the **.ERR1** error message appears on the screen and in the listing file. Like other error conditions occurring during pass one, the error generated by **.ERR1** does not cause the assembly to fail.

## Example

This example ensures that you define either the DOS or the OS2 symbol. If you define neither, the assembler assembles the nested **ELSE** condition and produces an error message. The **.ERR** directive causes an error on each pass.

```
IFDEF DOS  
    .  
    .  
    .  
ELSE  
    IFDEF OS2  
        .  
        .  
        .  
    ELSE  
        .ERR  
    ENDIF  
ENDIF
```

---

# .ERRB/.ERRNB (Error if Argument Blank/Non-Blank)

The **.ERRB** and **.ERRNB** directives test the given *Text-Argument*.

## Syntax

```
.ERRB Text-Argument  
    OR  
.ERRNB Text-Argument
```

## Remarks

If *Text-Argument* is blank, the **.ERRB** directive produces an error. If *Text-Argument* is not blank, **.ERRNB** produces an error.

You can test for the existence of parameters by using these directives within macros.

## Example

In this example, the directives ensure that only one argument is passed to the macro. If no argument is passed to the macro, the **.ERRB** directive produces an error. If more than one argument is passed, the **.ERRNB** directive produces an error.

```
WORK    MACRO    REALARG,TESTARG
        .ERRB    <REALARG>    ;; Error if no parameters
        .ERRNB   <TESTARG>    ;; Error if more than one parameter
        .
        .
        .
        ENDM
```

---

## .ERRDEF/.ERRNDEF (Error if Symbol Defined/Not Defined)

The **.ERRDEF** and **.ERRNDEF** directives test whether a symbol has been defined.

### Syntax

```
.ERRDEF Identifier
or
.ERRNDEF Identifier
```

### Remarks

If *Identifier* is defined as a label, a variable, or a symbol, the **.ERRDEF** directive produces an error. If you have not defined *Identifier*, **.ERRNDEF** produces an error. When *Identifier* is a forward reference, the assembler considers it undefined on the first pass and defined on the second pass.

### Example

In this example, **.ERRDEF** ensures that **SYMBOL** is not defined before entering the blocks, and **.ERRNDEF** ensures that you defined **SYMBOL** somewhere within the blocks.

```
.ERRDEF    SYMBOL
IFDEF     CONFIG1
    .
    .    SYMBOL EQU 0
    .
ENDIF
IFDEF     CONFIG2
    .
    .    SYMBOL EQU 1
    .
ENDIF
.ERRNDEF   SYMBOL
```

---

## .ERRDIF/.ERRDIFI (Error if Arguments are Different)

The **.ERRDIF** and **.ERRDIFI** directives generate an assembler error if the two *Text-Argument*s are different.

### Syntax

```
.ERRDIF Text-Argument-1, Text-Argument-2
or
.ERRDIFI Text-Argument-1, Text-Argument-2
```

## Remarks

The **.ERRDIF** directive performs a case-sensitive comparison, and the **.ERRDIFI** directive performs a case-insensitive comparison.

## Example

In this example, a check is made to verify that the currently opened segment is **\_TEXT**. This helps to insure that the macro is used only from within the default near code segment, and not from a program with a memory model that uses far code pointers (MEDIUM, LARGE, or HUGE).

```
RETURN MACRO
    ;; Use the expansion operator (%) to resolve @CurSeg equate
    % .errdif <_TEXT>, <@CurSeg>    ;; Must be in near .CODE segment
    RETN                            ;; Force a near return
ENDM
```

# .ERRE/.ERRNZ (Error if Expression False/True)

The **.ERRE** and **.ERRNZ** directives test the value of an *Expression*.

## Syntax

```
.ERRE Expression
or
.ERRNZ Expression
```

## Remarks

If the *Expression* evaluates to be false (zero), the **.ERRE** directive produces an error. If the *Expression* evaluates to be true (not zero), the **.ERRNZ** directive produces an error. The *Expression* must evaluate to an absolute value and cannot contain forward references.

## Example

In this example, **.ERRE** checks the boundaries of a parameter that the program passes to the macro **BUFFER**. If count is less than or equal to 128, the expression that the directive tests is true (not zero) and the directive produces no error. If **COUNT** is greater than 128, the expression is false (zero) and the directive produces an error.

```
BUFFER MACRO COUNT, BNAME
    .ERRE COUNT LE 128
    BNAME DB COUNT DUP (0) ;; Reserve memory, but no more than 128 bytes
ENDM

BUFFER 128, BUF1 ; Data reserved - no error
BUFFER 129, BUF2 ; Error produced
```

# .ERRIDN/.ERRIDNI (Error if Arguments are Identical)

The **.ERRIDN** and **.ERRIDNI** directives generate an assembly error if the two *Text-Argument*s are identical.

## Syntax

```
.ERRIDN Text-Argument-1, Text-Argument-2
or
.ERRIDNI Text-Argument-1, Text-Argument-2
```

## Remarks

The **.ERRIDN** directive performs a case-sensitive comparison, and the **.ERRIDNI** directive performs a case-insensitive comparison.

## Example

In this example, **.ERRIDN** protects against the passing the AX register as the second parameter, because the macro does not work if this happens. This example uses the **.ERRIDNI** directive since the macro needs to check for all possible spellings of the AX register.

```
ADDEM MACRO AD1,AD2,SUM
    .ERRIDNI <ax>,<AD2> ;; ERROR IF AD2 is ax
    MOV     AX,AD1      ;; Would overwrite if AD2 were AX
    ADD     AX,AD2
    MOV     SUM,AX      ;; SUM must be register or memory
ENDM
```

# Data Allocation

Data allocation statements allow you to reserve storage for your program data. To initiate a data allocation statement, an *Old-Style-Allocation-Directive* may be used, but in modes other than **M510** it is preferable to use a *Scalar-TypeName* or *UserDefined-TypeName*, which the assembler treats as a pseudo-directive. To introduce consistency into the descriptions, all such variations will be referred to as the *Allocation-TypeName*.

The *Allocation-TypeName* that you select determines the data-type of the allocated storage. An optional symbolic name may be associated with the storage, and the storage may also be initialized with specific values if so desired.

## Syntax

```
[Name] Allocation-TypeName Initializer [, Initializer ...]
```

Allocation-TypeName:

- Old-Style-Allocation-Directive*
- Scalar-TypeName*
- Record-TypeName*
- Structure-TypeName*
- Union-TypeName*
- Typedef-TypeName*

**Old-Style-Allocation-Directive:** one of

DB DW DD DF DQ DT

## Remarks

The various fields of the data allocation statement are described as follows:

**Name**

If the *Name* entry is present, it must be specified as a valid *Identifier* unique to the scope in which it

*Allocation-TypeName*

appears. If the allocation statement is assembled into an open segment, the assembler converts the identifier to a *Data-LabelName* to allow referencing the storage by a symbolic variable name. If the allocation statement is assembled into the body of a *STRUCT* or *UNION* type definition, then the assembler converts the identifier to a *Structure-FieldName* or *Union-FieldName*.

If the *Allocation-TypeName* is specified as a *Typedef-TypeName*, the assembler *resolves* it to its underlying data type to determine what type of initialization is to be performed.

If the *Allocation-TypeName* entry resolves to a *Scalar-TypeName* or a pointer to some other type, then the *Initializer* field must be specified using an expression syntax that can be resolved to a *Scalar-Initializer-ExpressionType*. See the following section on *Initialization of Scalar Types* for a full description of this topic.

If the *Allocation-TypeName* entry resolves to a *Record-TypeName*, *Structure-TypeName*, or *Union-TypeName*, then the *Initializer* field must be specified using the *Compound-Initializer* syntax. See the following section on *Initialization of Aggregate Types* for a full description of this topic.

If the *Allocation-TypeName* entry resolves to an array of any other type, then the *Initializer* field must be specified using the *Compound-Initializer* syntax. See the following section on *Initialization of Vector Types* for a full description of this topic.

*Initializer*

Each *Initializer* entry is an *Expression* that must resolve to an *Initializer-ExpressionType* appropriate for the type of data described by the *Allocation-TypeName* field.

Each *Initializer* entry may also be duplicated by making it the operand of a *Duplicative-Expression*. When assembling in *ALP* mode however, the *DUP* operator is considered obsolete and its use is discouraged. Instead, a *Typedef-TypeName* associated with the declaration of a true array should be used in the *Allocation-TypeName* field along with the appropriate compound initializer.

# Initialization of Scalar Types

A scalar data item represents a numeric quantity that may be increased or decreased in magnitude as a single unit. Thus, an *Initializer* expression for a scalar data item must be coded such that it resolves to a single scalar value. See the section on *Scalar-Initializer-ExpressionType* for the syntax and semantics of such expressions.

The old-style allocation directives (DB, DW, DD, DF, DQ, and DT) are supported in all assembler emulation modes, but for modes other than *M510*, the *Scalar-TypeName* keywords should be used instead.

When the *Scalar-TypeName* keywords are used instead of the old-style allocation directives, the assembler has full knowledge of the data types of the variables being created. This allows the assembler to make more intelligent code generation decisions, and it enables the assembler to correctly describe the variable in the symbolic debugging information that it generates for the source level debugger. *Scalar-TypeName*s may not be used as allocation directives in the *M510* mode.

To allocate an uninitialized scalar data item, use the *Indeterminate-Value-Alias* (\$) in the *Initializer* field.

Type Name Data Type	Initializer Description
DB, BYTE, Allocates 8-bit or SBYTE (byte) values.	Each <i>Initializer</i> must be in the range from 0 to 255 (unsigned) for a DB or

		BYTE directive, and from -128 to 127 (signed) for a SBYTE directive.
DW, WORD, or SWORD	Allocates 16-bit (word) values.	Each <i>Initializer</i> must be in the range from 0 to 65535 (unsigned) for a DW or WORD directive, and from -32768 to 32767 (signed) for a SWORD directive.
DD, DWORD, or SDWORD	Allocates 32-bit (double-word) values.	If the <i>Initializer</i> is an integer, each must be in the range from 0 to 4,294,967,295 (unsigned) for a DD or DWORD directive, and from -2,147,483,648 to 2,147,483,647 (signed) for a SDWORD directive. If the DD directive is being used, an <i>Initializer</i> may also resolve to a 32-bit <a href="#">Floating-Point-ExpressionType</a> .
DF or FWORD	Allocates 48-bit (6-byte far-word) values.	Each <i>Initializer</i> typically specifies the full address of a 32-bit far code or data label, but normal 32-bit integer values may also be used. The processor does not support 48-bit integer operations, thus the assembler does support 48-bit integer precision when initializing such variables. These directives are typically only useful for defining pointer variables for use on 32-bit processors.
DQ or QWORD	Allocates 64-bit (quad-word) values.	Both DQ and QWORD allow an integer <i>Initializer</i> with 64-bit (8-byte) precision. If the DQ directive is being used, the <i>Initializer</i> field may resolve to a 64-bit <a href="#">Floating-Point-ExpressionType</a> .
DT or TBYTE	Allocates 80-bit (10-byte) values	Both DT and TBYTE allow an integer <i>Initializer</i> with 80-bit (10-byte) precision. If the DT directive is being used, the <i>Initializer</i> field may resolve to a 80-bit <a href="#">Floating-Point-ExpressionType</a> .
REAL4, REAL8, or REAL10	Allocates real (floating-point) values of a specific size (4 bytes, 8 bytes, or 10 bytes).	Each <i>Initializer</i> must resolve to a <a href="#">Floating-Point-ExpressionType</a> . The assembler converts the floating-point literal to the IEEE format appropriate for the type of variable being allocated.

## Examples

Here are some examples of scalar initialization:

```
; Allocate some integer variables
uint8    BYTE    0, 255          ; min, max values for unsigned byte
sint8    SBYTE   -128, 127       ; min, max values for signed byte
USHORT_T  TYPEDEF WORD          ; Define a typedef alias for WORD
ushort   USHORT_T 0, 0FFFFh      ; and use it as allocation type name

; Some things to know about string-literal initializers
char      BYTE    "a"            ; a single BYTE value (061h)
is_int    WORD     "ab"          ; a single WORD value (06162h)
this_too  DWORD    "abcd"        ; a single DWORD value (061626364h)
too_long  WORD     "abcd"        ; error, expression too big for a word
string    BYTE     "string", 0    ; but strings can allocate many bytes

; Integers, pointers, and old-style initializations
PDWORD_T  TYPEDEF PTR DWORD      ; First, define a pointer type
ulong     DWORD    0, 0FFFFFFFh  ; min, max values for unsigned dword
pulong    PDWORD_T OFFSET ulong  ; pointer to the ulong variable
old_style DD      1.314          ; old style, floats are accepted
new_int   SDWORD   1.314          ; new style, error-must use integers
new_real  REAL4    1314           ; new style, error-must use floats

; Allocate some real numbers using decimal floating-point literals
float_f   REAL4    123.45         ; 4-byte IEEE real
double_f  REAL8    98.7654E1      ; 8-byte IEEE real
```

```

longdbl_f  REAL10    1000.0E-2           ;10-byte IEEE real

; The same values using hexadecimal floating-point literals
float_h    REAL4    42F6E666r           ; 4-byte IEEE real
double_h   REAL8    408EDD3B645A1CACr    ; 8-byte IEEE real
longdbl_h   REAL10   4002A000000000000000r ;10-byte IEEE real

```

## Initialization of Aggregate Types

An aggregate data item is a collection of one or more sub-items of possibly dissimilar types that are allocated, initialized, and treated as a single unit. The sub-items usually have unique names, and their positions relative to other sub-items is significant. The assembler provides the ability to define aggregate types through use of the [RECORD](#), [STRUCT](#), and [UNION](#) directives.

Initialization of an aggregate data item requires a programming notation that isolates the entire aggregate from surroundings constructs, and denotes the position of each sub-item within the aggregate. The syntax for this construct is as follows:

```

Aggregate-Initializer :
    { [Initializer-List] }
    < [Initializer-List] >

Initializer-List :
    Initializer-Item
    Initializer-List, [LineBreak] Initializer-Item

Initializer-Item :
    [Scalar-Initializer]
    [Aggregate-Initializer]
    [Array-Initializer]

```

The syntax requires that an *Aggregate-Initializer* be enclosed in an outer set of braces or angle brackets, but the *Initializer-List* or individual comma-separated *Initializer-Items* may be left unspecified, in which case a default initializer value is used. Commas are used to denote the position of each sub-item within the entire aggregate, and nested initializers are allowed to accommodate imbedded occurrences of other aggregates (or vector types, which share the same initializer syntax).

When initializing an instance of a union, the assembler only allows an initializer to be specified for the first field defined in the union type.

### Examples

Here are some examples of aggregate initialization:

```

YES    equ 1
NO     equ 0
MAYBE  equ -1

BOOL_T typedef sbyte

IDEAS_T struct
    sanctum  BOOL_T ?           ; For scalar data, use the ? operator
    peace   BOOL_T ?           ;   to request an uninitialized value.
    pilzner  BOOL_T ?
IDEAS_T ends

PROBLEM_T struct
    work    BOOL_T YES         ; Establish default initial values that
    car     BOOL_T NO          ;   can be inherited when an instance of
    house   BOOL_T MAYBE       ;   the structure is allocated
PROBLEM_T ends

SOLUTION_T struct
    fixing  PROBLEM_T {}       ; Outermost set of braces required even
    IDEAS_T {}                 ;   with unspecified (default) initializers
SOLUTION_T ends

DATA segment
    ProblemWith PROBLEM_T { NO, , MAYBE }           ; First-level structure
    ThinkOf     SOLUTION_T { { YES, YES, YES },     ; Initializer syntax for
                                { NO, NO, NO } }      ;   imbedded structures
DATA ends

```

```

CODE    segment
        assume ds:DATA
        mov al, NO
        or  al, ProblemWith.work
        or  al, ProblemWith.car
        or  al, ProblemWith.house
        jz  exit
        mov ThinkOf.fixing.work, NO      ; References to named fields in
        mov ThinkOf.fixing.car, NO       ; imbedded structures must be
        mov ThinkOf.fixing.house, NO     ; fully qualified.
exit:    mov ThinkOf.pilzner, YES         ; Reference to "promoted" field
        ret
CODE    ends
end

```

## Initialization of Vector Types

A vector data item is a linear collection of one or more sub-items of identical type that are allocated, initialized, and treated as a single unit. A vector (more commonly referred to as an *array*) is defined to have a specific number of items *n*, which are numbered from 0 to *n - 1* and occupy a contiguous area of allocated storage. The items in the vector may be of any type, possibly even other vectors (commonly known as a *multi-dimensional array*). The assembler provides the ability to define vector types through the use of the standard *Type-Declaration* syntax.

The syntax required to initialize a vector is similar to that used for an aggregate data type, and is as follows:

```

Array-Initializer :
    { [Initializer-List] }
    < [Initializer-List] >

Initializer-List :
    Initializer-Item
    Initializer-List, [LineBreak] Initializer-Item

Initializer-Item :
    [Scalar-Initializer]
    [Aggregate-Initializer]
    [Array-Initializer]

```

The syntax requires that an *Array-Initializer* be enclosed in an outer set of braces or angle brackets, but the *Initializer-List* or individual comma-separated *Initializer-Items* may be left unspecified, in which case a default initializer value is used. Commas are used to denote the position of each sub-item within the entire array, and nested initializers are allowed to accomodate imbedded occurrences of other arrays (or aggregate types, which share the same initializer syntax).

### Examples

Here are some examples of vector initialization:

```

; Data structures to define a "computer" data type

TRUE      equ      1
FALSE     equ      0
MB         equ      1024                ; Megabytes

BOOL_T     typedef  BYTE                ; true or false value
INCHES_T   typedef  BYTE                ; number of inches
MONITOR_T  typedef  INCHES_T            ; size of monitor in inches
KEYBOARD_T typedef  BOOL_T              ; is a keyboard installed?
MOUSE_T     typedef  BOOL_T              ; is a mouse installed?
KBYTES_T   typedef  WORD                ; number of kilobytes
MBYTES_T    typedef  WORD                ; number of megabytes
FPRESENT_T typedef  BOOL_T[2]           ; up to two floppies installed
FSIZE_T     typedef  KBYTES_T[2]        ; how big they are
DPRESENT_T  typedef  BOOL_T[4]          ; up to four hardfiles installed
DSIZE_T     typedef  MBYTES_T[4]        ; how big they are
RAM_T       typedef  DWORD              ; how much memory we have
NAME_T      typedef  BYTE[64]           ; what we call the system

FLOPPIES_T struct
DriveCount FPRESENT_T { TRUE, FALSE } ; assume one floppy installed

```

```

DriveSize  FSIZE_T      { 360, 0 }          ; assume 360KB in size :-)
FLOPPIES_T ends

DRIVES_T   struct
DriveCount DPRESNT_T { TRUE, FALSE, FALSE, FALSE } ; one drive installed
DriveSize  DSIZE_T    { 20, 0, 0, 0 }           ; 20MB in size (!)
DRIVES_T   ends

COMPUTER_T struct
Monitor    MONITOR_T   14                     ; Assume a 14 inch monitor
Keyboard   BOOL_T       TRUE                   ; We have a keyboard
Mouse      BOOL_T       FALSE                  ; but no mouse
Memory     RAM_T         640                   ; Assume 640KB
Floppies   FLOPPIES_T   {}                    ; Go with the defaults
HardFiles  DRIVES_T      {}                    ; Go with the defaults
ModelName  NAME_T        {}                    ; No default name
COMPUTER_T ends

DATA segment
Circa1997  COMPUTER_T \                        ; initializer begins on next line
{ 17,                                           ; of course, we have a 17" monitor
  TRUE, TRUE,                                  ; a keyboard and a mouse
  32 * MB,                                     ; 32 Megabytes of ram
  { { },                                       ; still one floppy
    { 1440 } },                               ; but it has a 1.2 MB capacity
  { { , TRUE, TRUE },                         ; also have second and third hardfiles
    { 512, 1024, 4096 } },                   ; 512MB, 1 GIG, and 4 GIG
  { "Spiffatron 9000", 10, 13,               ; with a fancy system name
    "Acme Computers", 10, 13, 0 } }
DATA ends
end

```

## Intermodule Linkage

To use symbols and procedures in more than one module, ALP must recognize shared data as global to all modules. ALP provides directives to simplify data sharing and a high-level interface to multiple-module programming. With these directives, you can define shared symbols and refer to them from other modules.

This section describes the following intermodule linkage directives:

```

COMM
END
EXTERN/EXTRN
EXTERNDEF
INCLUDELIB
NAME
PUBLIC

```

## COMM (Declare Communal Variable)

Declares an uninitialized *common* or *communal* variable that is allocated by the linker.

### Syntax

```

COMM [Language-Name] [Distance] Name [Count]:TypeName[:Size] [, ...]

```

### Remarks

The arguments to the **COMM** directive are as follows:

*Language-Name*

Optional parameter that determines how *Name* is spelled when written to the object file. Used when interfacing with routines written in high-level languages. If not specified, the language defaults to the value set by [.MODEL](#) or [OPTION LANGUAGE](#).

*Distance*

One of **NEAR** or **FAR**; determines the distance of allocated variable. If not specified, the current memory model determines the distance. The default is **NEAR** if no memory model is active.

*Name*

The name of the variable to be allocated by the linker. This field is required.

*[Count]*

Optional; if specified, this parameter must be surrounded by square brackets. The *Count* parameter can be thought of as a major (row) array dimension. It defaults to 1 if not specified.

*TypeName*

Required parameter that specifies the type of the variable being allocated. It must be a single keyword or identifier that specifies a [Distance-TypeName](#), [Scalar-TypeName](#), or [UserDefined-TypeName](#).

*Size*

*Size* is an optional parameter that can be thought of as a minor (column) array dimension. It defaults to 1 if not specified.

Communal variables are allocated by the linker. When the linker combines object modules together, all instances of an identically-named communal variable are merged into a single instance (union), and are uninitialized.

The allocated size of a communal variable is the largest size requested by all encountered references.

The allocation order with respect to the addresses of other global symbols is undefined; an application must not depend on the address of a communal variable being less than or greater than that of another global symbol.

A variable allocated with the **COMM** directive need not be declared in all referencing modules as communal; the linker matches all [EXTERN/EXTRN](#) references with that of the communal variable. Similarly, a variable allocated in one module with the [PUBLIC](#) directive may be declared in other modules as communal.

Since communal variables cannot be initialized and their address positions cannot be compared, use of the **COMM** directive is discouraged. The [EXTERNDEF](#) directive should be used instead.

---

## END (Define End of Module and Entry Point)

The **END** directive has two functions:

- Identifies the end of the source program.
- Identifies the symbol that is the name of the entry point (through the [Expression](#) on the **END** directive)

### Syntax

```
END [Expression]
```

### Remarks

All source files must have the **END** directive as the last statement. Any lines following the **END** statement are ignored by the assembler.

When the linker builds an application program from one or more object modules, it needs to know where the entry point is for the operating system to pass initial control. If you do not specify an entry point, none is assumed. Only one module can identify a label as the entry point by specifying that label on its **END** statement. Any module not defining an operating system entry point must not have an entry point identified on its **END** statement. If you fail to define an entry point for the main module, your program may not be able to initialize correctly. It will assemble and link without error, but it cannot run.

### Example

The following example is the **END** statement for the section of code that starts with the name **BEGIN**.

END BEGIN

-----

## EXTERN/EXTRN (Declare External Identifier)

The **EXTERN** directive specifies a declaration for the external symbol *Name* so that it may be referred to within this module. The actual definition for the symbol occurs in some other module, and the linker resolves all such external declarations to a single definition for *Name*.

### Syntax

```
EXTERN [Language-Name] Name [(Default-Resolution)] :Type [, ...]
```

Where *Type* is one of:

- **ABS**
- *Type-Declaration*

### Remarks

The obsolete spelling for the **EXTERN** directive is **EXTRN**.

The external source module that defines the symbol must give it public visibility in the corresponding object module, which is accomplished in assembler language by declaring it with the **COMM** directive, defining the symbol in association with an **EXTERND** or **PUBLIC** directive, or by specifying the **PUBLIC** or **EXPORT** attributes in a **PROC** directive.

If the **EXTERN** directive is given within a segment, the assembler assumes that the symbol is located within that segment. If the segment is not known, place the **EXTERN** directive outside all segments and either use an explicit segment prefix or an **ASSUME** directive.

A *Type* value of **ABS** indicates that *Name* is an externally-defined constant value. Local references to *Name* are treated as immediate values having an *Operand Size* equal to the *Address Size* of the segment containing the reference.

**Note:** If the *Type* of **EXTERN** is **ABS**, it may not be used anywhere in this module where conversion to an immediate value of type **BYTE** is required. Additionally, the defining module must define the value as a constant symbol.

**For example:**

```
FOO    EQU    5
PUBLIC FOO
```

Use of the *(default\_resolution)* syntax declares the external symbol *Name* to be a "weak" symbol, in which case the linker will pair all such declarations with the symbol *default\_resolution* unless a standard "strong" public definition for *Name* is encountered during the link.

### Example

IN THE SAME SEGMENT

IN MODULE 1:

```
cseg segment
public tagn
.
.
.
tagn:
.
.
.
cseg ends
```

IN ANOTHER SEGMENT

IN MODULE 1:

```
csega segment
public tagf
.
.
.
tagf:
.
.
.
csega ends
```

<pre>IN MODULE 2:  cseg segment extern tag:near . . . jmp tagn cseg ends</pre>	<pre>IN MODULE 2:  extern tagf:far csegb segment . . . jmp tagf csegb ends</pre>
--------------------------------------------------------------------------------	----------------------------------------------------------------------------------

-----

## EXTERNDEF (Declare Global Identifier)

The **EXTERNDEF** directive combines the functionality of the [EXTERN/EXTRN](#) and [PUBLIC](#) directives. It provides a uniform way to declare global symbols that are to be shared across multiple modules.

### Syntax

```
EXTERNDEF [Language-Name] Name:Type [, ...]
```

Where *Type* is one of:

- **ABS**
- *Type-Declaration*

### Remarks

A symbol declared with **EXTERNDEF** is treated as [PUBLIC](#) if a definition for the symbol is encountered during the assembly, otherwise the symbol is assumed to be defined in another module and is treated as if it were declared with the [EXTERN/EXTRN](#) directive.

### Example

The following example shows how a declaration for the **ReturnCode** symbol can be shared between two modules (Main.asm and FileErr.asm) by way of a common header file (ErrNum.inc):

```
; -----
; ErrNum.inc

RETCODE_T typedef DWORD

RC_NoError      equ 0
RC_FileNotFound equ 1
RC_SystemError  equ 3

EXTERNDEF ReturnCode:RETCODE_T      ; declaration

; -----
; FileErr.asm
.386
.MODEL FLAT
INCLUDE ErrNum.inc                  ; bring in error number definitions
                                   ; and declaration for ReturnCode

.CODE
; Tell the user about the file error,
; then make sure the program has a non-zero exit status
FileError proc
    ...
    mov ReturnCode, RC_FileNotFound
    ret
FileError endp
end
```

```

; -----
; Main.asm
.386
.MODEL FLAT
INCLUDE ErrNum.inc           ; bring in error number definitions
                             ; and declaration for ReturnCode
EXTERNDEF FileError:PROC     ; This could be in a common header too

.DATA
ReturnCode RETCODE_T RC_NoError ; actual definition of ReturnCode

.CODE
Main proc
    ...
    call FileError           ; hypothetical error condition
    ...
    mov eax, ReturnCode      ; load the exit status
    call Exit                ; and shutdown the program
Main endp
end Main

```

## INCLUDELIB (Pass Library Name to Linker through Object File)

The **INCLUDELIB** directive is used to inform the linker that a library file of a given name is to be used when attempting to resolve external references declared by this module.

### Syntax

```
INCLUDELIB FileName
```

### Remarks

The *FileName* argument is parsed as a contiguous string of arbitrary characters, and should constitute a file name that is valid in the context where it will be used. The *FileName* should be coded as a *<text-literal>* if it is to contain embedded spaces or other special characters.

The assembler emits a special record into the object file which contains the string of characters given by the *FileName* entry. This record instructs the linker to include the named library file in its list of libraries to be searched during the process of resolving external references. The assembler attaches no other meaning to the object file record, and it is up to the linker to interpret the file name for any special meaning (such as search path information, file name extension, and so on).

Use of this directive avoids the need to explicitly reference the library name in a linker invocation parameter, and helps to avoid the problems that can arise when such parameters are specified incorrectly.

### Example

```
INCLUDELIB OS2386.LIB
```

## NAME (Specify Module Name)

The **NAME** directive assigns a module a name.

## Syntax

```
NAME module-name
```

## Remarks

The **NAME** directive is ignored; it is provided for backward compatibility with other assemblers.

-----

# PUBLIC (Make Symbol Visible to Other Modules)

The **PUBLIC** directive makes defined symbols available to other programs that are to be linked. The information referred to by the **PUBLIC** directive is passed to the linker.

## Syntax

```
PUBLIC [Language-Name] Identifier[,...]
```

## Remarks

*Identifier* can be a variable or a label (including **PROC** labels). Register names and any symbols defined by **EQU** or = to floating-point numbers or integers larger than 4 bytes are incorrect entries.

## Example

```
                PUBLIC  GETINFO  ;Make GETINFO visible to linker
GETINFO PROC    FAR
                PUSH    BP        ;Save caller's register
                MOV     BP,SP      ;Get address of parameters
                                ;BODY OF SUBROUTINE
                POP     BP        ;restore caller's register
                RET      ;return to caller
GETINFO ENDP
```

-----

# Listing Control

ALP creates an assembler listing of your source file whenever you use a related source code directive or specify the **+FI** option on the ALP command line.

The assembler listing contains:

- Cumulative Listing Line Number
- Individual Source File Line Number
- Macro Expansion Line Number
- Macro Definition Line Number
- Macro Expansion Indentation Level
- Macro Expansion Nesting Level
- Include File Nesting Level
- Conditional Assembly Nesting Level
- True or False Conditional Flag
- Location Counter Offset Value

- Generated Machine Code Data
- Source Line Data

If requested (via the `+Ls` command line option) a symbol table listing is produced that shows the names and values of all of the user-defined identifiers created during the assembly. The values of certain predefined identifiers are also show in the symbol table listing.

The symbol table listing is divided into the following categories:

- Macro Names
- Text Equate Names
- Structures/Union Type Names
- Orphaned Structure Fields
- Record Type Names
- Typedef Type Names
- Group Names
- Segment Names
- Numeric Equate Names
- Code Label Names
- Procedure Names
- Variable Names

ALP places the symbol table listing at the end of the listing output. ALP lists only the types of symbols encountered in the program. For example, if your program does not define any macros, the **Macro Names** section is omitted from the listing output.

This section describes the following listing control directives:

```
%BIN
.CREF
.LALL
.LFCOND
.LIST
.LISTALL
.LISTIF
.LISTMACRO
.LISTMACROALL
.NOCREF
.NOLIST
.NOLISTIF
.NOLISTMACRO
PAGE
.SALL
.SFCOND
SUBTITLE
SUBTTL
.TFCOND
TITLE
.XALL
.XCREF
.XLIST
```

---

## %BIN (Set Listing Width for Object Code Field)

Sets the width of the object code field in the listing file to *size* columns.

### Syntax

```
%BIN size
```

---

## .CREF/.XCREF (Control Symbol Cross Referencing)

The output of the cross-reference information is controlled by these directives. The default condition is the **.CREF** directive. When the assembler finds a **.XCREF** directive, cross-reference information results in no output until the assembler finds

**Note:** The assembler does not produce cross-referencing information. These directives are provided for source file compatibility with other assemblers.

### Syntax

```
.CREF
    or
.XCREF [[operand [, ...]]
```

### Remarks

The **.XCREF** directive can have an optional operand consisting of a list of one or more variable names suppressed in the cross-reference listing.

---

## .LFCOND (List False Conditionals)

You use the **.LFCOND** (List False Conditionals) directive to list conditional blocks that are evaluated as false.

### Syntax

```
.LFCOND
```

### Remarks

Equivalent to the **.LISTIF** directive.

**.LFCOND** does not have an operand. You can end this state either by issuing **.TFCOND**, which reverts to the default state concerning listing of false conditionals (but with the default state redefined as being in the opposite state,) or by issuing the **.SFCOND**, which suppresses the listing of false conditionals.

The assembler does not print false conditionals within macros when **.LALL** is set.

---

## .LIST/.XLIST (Control Listing File Output)

These two directives control output to the listing file.

### Syntax

```
.LIST
    or
.XLIST
```

### Remarks

If a listing is not being created, these directives have no effect. The **.LIST** is the default condition. When the assembler finds an **.XLIST**, the assembler does not list the source and the object code until it finds a **.LIST** directive.

---

## .LISTALL (List All Statements)

Starts the listing of all statements.

### Syntax

```
.LISTALL
```

### Remarks

Equivalent to the combination of **.LIST**, **.LISTIF**, and **.LISTMACROALL**.

---

## .LISTIF (List False Conditionals)

Starts the listing of all statements, including those in false conditional blocks.

### Syntax

```
.LISTIF
```

### Remarks

Equivalent to the combination of **.LIST**, **.LISTIF**, and **.LISTMACROALL**.

---

## .LISTMACRO/.XALL (List Code and Data Statements in Macros)

Starts listing of only those statements that generate code or data when processing macro expansions.

### Syntax

```
.LISTMACRO  
or  
.XALL
```

### Remarks

ALP does not support this mode; it is provided for compatibility with other assemblers.

---

## .LISTMACROALL/.LALL (List All Statements in Macros)

Starts listing of all statements when processing macros expansions.

## Syntax

```
.LISTMACROALL  
  OR  
.LALL
```

-----

## .NOCREF (Suppress Symbol Cross Referencing)

Suppresses the listing of symbols in the symbol table and cross-referencing output.

**Note:** The assembler does not produce cross-referencing information. This directive is provided for source file compatibility with other assemblers.

## Syntax

```
.NOCREF [name[,name] ...]
```

## Remarks

If names are specified, only the given names are suppressed. Same as **.XCREF**.

-----

## .NOLIST (Suppress List Output)

Suppresses program listing.

## Syntax

```
.NOLIST
```

## Remarks

Same as **.XLIST**.

-----

## .NOLISTIF (Do Not List False Conditionals)

Suppresses listing of conditional blocks whose condition evaluates to false (0).

## Syntax

**.NOLISTIF**

#### Remarks

This is the default. Same as **.SFCOND**.

---

## .NOLISTMACRO (Do Not List Macro Expansions)

Suppresses listing of macro expansion.

#### Syntax

**.NOLISTMACRO**

#### Remarks

Same as **.SALL**.

This is the default setting for ALP.

---

## PAGE (Control Listing Page Length and Width)

The **PAGE** directive controls the length and width of each listing page. Place the **PAGE** directive in the source file to control the format of the listing file produced during assembly.

#### Syntax

```
PAGE [operand-1] [, operand-2]  
      or  
PAGE +
```

#### Remarks

Using **PAGE +** or the **PAGE** directive without an operand entries causes the printer to go to the top of the page and increases the page number by 1. The assembler normally takes this action only when a page is full.

The *operand-1* entry specifies the actual number of lines that can be physically printed on the page; the default value is 66.

Use the *operand-2* entry to control the width of the page. The page width without a specified number is 132.

**Note:** The **PAGE** directive does not set the printer to the desired line width. For proper formatting of the listing, initialize the printer to operate at a corresponding line width before printing the listing file.

---

## SUBTITLE/SUBTTL (Specify Listing Page Subtitle)

Defines the subtitle displayed in the user area of each page in the listing output.

#### Syntax

```
SUBTITLE text
or
SUBTTL text
```

---

## .TFCOND (Toggle Listing of False Conditionals)

Toggles listing of false conditional blocks.

### Syntax

```
.TFCOND
```

---

## TITLE (Specify Listing Page Title)

Defines the title displayed in the user area of each page in the listing output.

### Syntax

```
TITLE text
```

---

## Procedure Control

Procedure control directives allow you to organize your code into procedures. The **PROC** and **ENDP** directives mark the beginning and end of a procedure. Also, **PROC** can automatically:

- Preserve higher register values that should not change but that the procedure might otherwise alter
- Set up a local stack pointer, so that you can access parameters and local variables placed on the stack
- Adjust the stack when the procedure ends

This section describes the following procedure control directives:

```
PROC
LOCAL
ENDP
```

---

## PROC (Identify Code Procedure)

The **PROC** directive identifies a block of code. By dividing the code into blocks, each of which performs a distinct function, you can clarify the overall function of the complete module.

The **PROC** directive also identifies the procedure *distance* to help insure that the assembler generates the appropriate instructions for calling and returning from the procedure while maintaining the integrity of the run-time stack.

## Syntax

```
Procedure-Name PROC [Attributes] [Register-List] [Parameter-List]
.
.
.
RET [Constant]
.
.
.
Procedure-Name ENDP
```

Refer to the following sections for descriptions of the optional arguments to the **PROC** directive:

- [Attributes](#)
- [Register-List](#)
- [Parameter-List](#)

## Remarks

You can execute the block of code identified by the **PROC** directive in-line, jump to it, or start it with a **CALL** instruction. If the **PROC** is called from code that has another **ASSUME CS** value, you must use the appropriate **FAR**, **FAR16**, or **FAR32** *distance attribute*.

The **NEAR** attribute causes any **RET** instruction coded within the procedure to be an intra-segment return that pops a return *offset* from the stack. You can call a **NEAR** subroutine only from the same segment. However, the **FAR** attribute causes **RET** to be an inter-segment return that pops both a return *offset* and a *segment base* from the stack. You can call a **FAR** subroutine from any segment; a **FAR** subroutine is usually called from a segment other than the one containing the subroutine.

## Example

In this example, the **Near\_Name** subroutine is called by the **Far\_Name** subroutine.

```
Far_Name PUBLIC Far_Name
PROC FAR
CALL Near_Name
RET ;Pops return offset and seg base value
Far_Name ENDP

Near_Name PUBLIC Near_Name
PROC NEAR
.
.
.
RET ;pops only return offset
Near_Name ENDP
```

You can call the **Near\_Name** subroutine directly from a **NEAR** segment by using:

```
CALL Near_Name
```

A **FAR** segment can indirectly call the second subroutine by first calling the **Far\_Name** subroutine with:

```
CALL Far_Name
```

A **CALL** to a forward-referenced symbol assumes the symbol is **NEAR**. If that symbol is **FAR**, the **CALL** must have an override, for example:

# Attributes

The optional fields in the *Attributes* argument control how the procedure is defined.

## Syntax

[*Distance*] [*Language*] [*Visibility*]

## Remarks

The various *Attribute* fields are defined as follows:

### *Distance*

Determines the type of **CALL** instruction that should be used to invoke the procedure, and the type of **RET** instruction generated by the assembler. The default is **NEAR** if no **.MODEL** directive has been specified, or if the model has been set to **TINY**, **SMALL**, **COMPACT**, or **FLAT**. The default is **FAR** if the model has been set to **LARGE**, **MEDIUM**, or **HUGE**. If the programmer is using segments with mixed address sizes (**USE16** and **USE32**) on a 32-bit processor, then the **NEAR16**, **FAR16**, **NEAR32**, and **FAR32** keywords may also be used.

### *Language*

Determines the calling convention used by the procedure, and the naming convention used when writing the procedure name to the object file. The calling convention defines the layout of the stack frame upon entry to the procedure and how the stack frame is destroyed upon procedure exit. See the section on [Label/Names](#) for more information on language naming conventions.

With the **BASIC**, **FORTRAN**, and **PASCAL** calling conventions, the called procedure expects arguments to be pushed on the stack from left to right, causing the rightmost parameter to be at the lowest stack address and closest in proximity to the *frame pointer* (the BP or EBP register). With this arrangement, the called procedure always knows the exact amount of stack space used by the parameters, and is responsible for removing them from the stack with a **RET Constant** instruction when the procedure exits. Such procedures are unable to accept a variable number of arguments.

With the **C**, **STDCALL**, **SYSCALL**, and **OPTLINK** calling conventions, the called procedure expects arguments to be pushed on the stack from right to left, causing the leftmost parameter to be at the lowest stack address and closest in proximity to the *frame pointer* (the BP or EBP register). With this arrangement, the calling procedure is free to push additional arguments on the stack, and is responsible for restoring the stack after the called procedure returns (**STDCALL** requires the called procedure to restore the stack if a fixed number of arguments is being passed).

With the **OPTLINK** 32-bit calling convention (as defined by the IBM VisualAge C/C++ Compiler environment), up to three parameters will be passed in machine registers to the called procedure, provided they not larger than a DWORD in size. The EAX, EDX, and ECX registers (respectively) are used for this purpose. Stack space for the parameters is still allocated, but the parameter values are not actually copied onto the stack. Refer to the documentation for the IBM VisualAge C++ compiler for more information on the **OPTLINK** calling convention.

### *Visibility*

Determines if the procedure name is written to the object file as a global identifier, allowing it to be referenced by other modules. The allowable values are **PRIVATE**, **PUBLIC**, and **EXPORT**. If operating in **M510** mode and no **.MODEL** directive with a *Language-Name* has been specified, then the default visibility is **PRIVATE**. In all other situations, the default visibility is **PUBLIC** unless the default has been overridden by an **OPTION LANGUAGE** directive.

When the **PRIVATE** keyword is used, the procedure name is visible only within

the defining module at assembly-time. When the visibility is **PUBLIC**, the procedure name is made visible to other modules at link-time. The same is true of **EXPORT** visibility, but in this case the assembler emits a special record into the object file that causes the linker to also make the symbol visible as an exported entry point in the executable module, allowing it be called by other modules at program run-time.

---

## Register List

The optional *Register-List* defines those registers used in the body of the procedure that must be preserved on behalf of the caller. The assembler generates code to save these registers on the stack when the procedure is entered, and to restore them when the procedure exits.

### Syntax

```
USES Register [ Register...]
```

*Register:*

*16-Bit-Register*

*32-Bit-Register*

*Segment-Register*

### Remarks

When more than one register is specified, do not use commas to separate the register keywords; use blanks or tabs instead.

---

## Parameter List

The optional *Parameter-List* defines the parameters that the caller passes to the procedure on the run-time stack.

### Syntax

```
[, [LineBreak]] Parm-List
```

*Parm-List:*

*Parm-Spec* [, [LineBreak] *Parm-Spec* ...]

*Parm-Spec:*

*Parameter-Name* [: *Type*]

The introductory comma in front of the *Parm-List* is only required if a *LineBreak* is used to put the first *Parm-Spec* on the line following the **PROC** directive.

The optional *LineBreak* entry allows you to end a *Parm-Spec* entry with a comma, enter an optional *EndOfLine-Comment* followed by a physical *NewLine* character, then continue the *Parm-List* on the next line.

### Remarks

Each *Parameter-Name* is defined as a *Numeric-EquateName* that is visible only from within the body of the procedure. The value assigned to the parameter name is an expression that defines the parameter type and its location on the stack relative to the value of the *frame pointer* (the BP or EBP register). The assembler automatically calculates the correct offset value based upon the size of the parameter type.

The *Type* field is specified as a *Type-Declaration* and defines the data type associated with the *Parameter-Name*. If this field is omitted, the data type defaults to **WORD** if the procedure is defined within a **USE16** segment, and **DWORD** if the procedure is defined within a **USE32** segment.

The programmer can read from and store into the locations defined by the *Parm-Spec* entries as though they were regular named variables, but if the parameter names are to be combined in indexed expressions with other registers, the normal rules for specifying **BP**- and **EBP**-relative expressions must be followed.

## Example

This example defines a **ReadBuffer** procedure to accept four arguments passed on the stack.

```
.386                                ; Assemble for 32-bit processors
.model flat,syscall                 ; OS/2 programming model/calling convention

EXTERN DosRead:PROC                 ; OS/2 DosRead() API
INCLUDELIB os2386.lib               ; This lets us link to the API

.code                               ; Open the code segment

;-----
; Call operating system to read input into a buffer
;-----
ReadBuffer PROC,                    ; need comma to continue the PROC statement
    hFile:dword,                    ; parm 1: Read handle
    pBuffer:ptr byte,               ; parm 2: Address of input buffer
    cbRead:dword,                   ; parm 3: Size of input buffer
    pulActual:ptr dword             ; parm 4: Address of byte count from read

    ; set up to call the OS/2 DosRead entry point

    PUSH    pulActual               ; arg 4
    PUSH    cbRead                  ; arg 3
    PUSH    pBuffer                 ; arg 2
    PUSH    hFile                   ; arg 1
    CALL    DosRead                 ; Invoke syscall (SYSTEM) function
    ADD     ESP,DWORD * 4            ; Remove the four parameters we pushed
                                        ; onto the stack for the DosRead call
    RET
ReadBuffer    ENDP
```

# LOCAL (Define Local Procedure Variables)

The **LOCAL** directive defines local stack variables from within a code procedure.

## Syntax

```
LOCAL Local-Spec [, [LineBreak] Local-Spec ...]
```

### *Local-Spec:*

*Local-Name* [: *Type-Declaration*]

*Local-Name*[*Count*] [: *Type-Declaration*]

The optional *LineBreak* entry allows you to end a *Local-Spec* entry with a comma, enter an optional *EndOfLine-Comment* followed by a physical *NewLine* character, then continue with a new *Local-Spec* on the next line.

## Remarks

The **LOCAL** assembler directive can only appear within the body of a procedure. If used, the **LOCAL** directive(s) must immediately follow the **PROC** statement that encloses them, and they must appear before any instructions, code labels, or directives that modify the location counter. Multiple **LOCAL** directives may appear in succession.

Each *Local-Name* is defined as a *Numeric-EquateName* that is visible only from within the body of the procedure. The value assigned to the variable name is an expression that defines the type of the variable and its location on the stack relative to the value of the *frame pointer* (the BP or EBP register). The assembler reserves space on the stack for each local variable and automatically calculates their locations. After all *Local-Spec* entries have been processed, the assembler allocates the space by generating instructions to adjust the stack pointer. The assembler also generates

instructions to restore the state of the stack and frame pointers when the procedure exits.

The optional *[Count]* entry can be used to indicate that the variable is a simple "array" of values, where *Count* is a constant expression. If used, the square brackets surrounding the *Count* must be specified. Use of this notation is discouraged however, because it does not associate a "true array" data type with the variable, and cannot be viewed as such from within a symbolic debugger. ALP allows the variable to be associated with a "true array" data type through use of the native *Type-Declaration* syntax.

The *Type-Declaration* field specifies the data type to be associated with the *Local-Name*. If this field is omitted, the data type defaults to **WORD** if the procedure is defined within a **USE16** segment, and **DWORD** if the procedure is defined within a **USE32** segment.

## Example

```
; bootdrv.asm : Returns value of OS/2 boot drive as exit code
; assemble as : alp +Od bootdrv.asm
; link as      : link386 /de bootdrv;

        .386                                ; Assemble for 32-bit processors
        .model flat,syscall                 ; OS/2 flat model/calling convention
        .stack 4096

        EXTERN DosExit:PROC                ; OS/2 DosExit() API
        EXTERN DosQuerySysInfo:PROC        ; OS/2 DosQuerySysInfo() API
        INCLUDELIB os2386.lib              ; link with these routines

; These are values taken from OS/2 API headers. See the OS/2 Toolkit
; Control Program Programming Guide and Reference for more information.

EXIT_PROCESS EQU 1                        ; for DosExit
QSV_BOOT_DRIVE EQU 5                      ; For DosQuerySysInfo

ULONG     TYPEDEF DWORD                   ; use OS/2 type convention

        .code                               ; open code segment

main PROC
    LOCAL BootDrive:ULONG                 ; place to put value of boot drive

    ; Push parameters to DosQuerySysInfo onto the stack

    PUSH    sizeof BootDrive              ; arg 4: size of output buffer
    LEA     EAX,BootDrive                  ; arg 3: Address of buffer
    PUSH    EAX
    PUSH    QSV_BOOT_DRIVE                 ; arg 2: last ordinal value to return
    PUSH    QSV_BOOT_DRIVE                 ; arg 1: first ordinal, same as last
    CALL    DosQuerySysInfo                ; invoke API
    ADD     ESP,DWORD * 4                  ; remove the parameters from the stack

    CMP     EAX,0                          ; Did the API succeed?
    MOV     EAX,0                          ; if not, use zero as a return code
    JNZ     SomeKindOfError                ; and skip around to the exit logic
    MOV     EAX,BootDrive                  ; else, return the boot drive value
SomeKindOfError:
    push    EAX                            ; exit code
    push    EXIT_PROCESS                   ; terminates all threads
    call    DosExit                        ; exit to calling process
    RET                                         ; never executed
main ENDP

        END     main
```

---

## ENDP (Close a Procedure Definition Block)

Every procedure block opened with the **PROC** directive must be ended with the **ENDP** directive.

### Syntax

*procedure-name* **ENDP**

## Remarks

If the **ENDP** directive is not used with the **PROC** directive, an error occurs. An unmatched **ENDP** also causes an error.

**Note:** See the **PROC** directive in this chapter for more detail and examples of **ENDP** use.

## Example

```
PUSH    AX                ; Push third parameter
PUSH    BX                ; Push second parameter
PUSH    CX                ; Push first parameter
CALL    ADDUP             ; Call the procedure
ADD     SP,6              ; Bypass the pushed parameters
.
.
.
ADDUP   PROC    NEAR      ; Return address for near call
                                ; takes two bytes
                                PUSH    BP                ; Save base pointer - takes two more
                                ; so parameters start at 4th byte
                                MOV     BP,SP              ; Load stack into base pointer
                                MOV     AX,[BP+4]          ; Get first parameter
                                ; 4th byte above pointer
                                ADD     AX,[BP+6]          ; Get second parameter
                                ; 6th byte above pointer
                                ADD     AX,[BP+8]          ; Get third parameter
                                ; 8th byte above pointer
                                POP     BP                ; Restore base
                                RET                     ; Return
ADDUP   ENDP
```

In this example, three numbers are passed as parameters for the procedure **ADDUP**. Parameters are often passed to procedures by pushing them before the call so that the procedure can read them off the stack.

---

# Processor Control

ALP provides a set of directives for selecting processors and coprocessors. Once you select a processor, you must only use the instruction set available for that processor. The default is the 8086 processor. If you always want your code to run on this processor, you need not add any processor directives.

This section describes the following processor control directives:

.8086  
.8087  
.186  
.286  
.286P  
.287  
.386  
.386P  
.387  
.486  
.486P  
.586  
.586P  
.686  
.686P  
.MMX  
.NOMMX  
.SIMD  
.NOSIMD  
.XMM

---

## .8086 (Select 8086 Processor Instruction Set)

The **.8086** directive tells the assembler to recognize and assemble 8086 instructions. This directive assembles only 8086 and 8088 instructions (the 8088 instructions are identical to the 8086 instructions). ALP assembles 8086 instructions by default.

### Syntax

```
.8086
```

### Remarks

The **.8086** directive does not have an operand.

**Note:** The **.8086** directive does not end ALP 8087/80287 mode.

---

## .8087 (Select 8087 Coprocessor Instruction Set)

The **.8087** directive tells the assembler to recognize and assemble 8087 instructions and data formats. ALP assembles 8087 instructions by default.

### Syntax

```
.8087
```

### Remarks

The **.8087** directive does not have an operand.

---

## .186 (Select 80186 Processor Instruction Set)

The **.186** directive tells the assembler to recognize and assemble 8086 or 8088 instructions and the additional instructions for the 80186 microprocessor.

### Syntax

```
.186
```

### Remarks

The **.186** directive does not have an operand. Use it only for programs that run on an 80186 microprocessor.

---

## .286 (Select 80286 Processor Instruction Set)

Enables assembly of nonprivileged instructions for the 80286 processor. Disables assembly of instructions introduced with later processors. Also enables 80287 instructions.

#### Syntax

`.286`

---

## `.286P` (Select 80286 Processor Protected Mode Instruction Set)

The `.286P` directive tells the assembler to recognize and assemble the protected instructions of the 80286 in addition to the 8086, 8088, and nonprotected 80286 instructions.

#### Syntax

`.286P`

#### Remarks

The `.286P` directive does not have an operand. Use it only for programs run on an 80286 processor using both protected and nonprotected instructions.

---

## `.287` (Select 80287 Coprocessor Instruction Set)

The `.287` directive tells the assembler to recognize and assemble instructions for the 80287 floating point math coprocessor. The 80287 instruction set consists of all 8087 instructions, plus three additional instructions.

#### Syntax

`.287`

#### Remarks

The `.287` directive does not have an operand. Use it only for programs that have 80287 floating point instructions and run on an 80287 math coprocessor.

---

## `.386` (Select 80386 Processor Instruction Set)

Enables assembly of nonprivileged instructions for the 80386 processor. Disables assembly of instructions introduced with later processors. Also enables 80387 instructions.

#### Syntax

`.386`

---

## `.386P` (Select 80386 Processor Protected Mode Instruction Set)

Enables assembly of all instructions (including privileged) for the 80386P processor. Disables assembly of instructions introduced with later processors. Also enables 80387 instructions.

### Syntax

`.386P`

---

## `.387` (Select 80387 Coprocessor Instruction Set)

Enables assembly of instructions for the 80387 coprocessor.

### Syntax

`.387`

---

## `.486` (Select 80486 Processor Instruction Set)

Enables assembly of instructions for the 80486 processor. Also enables 80387 (and later) floating point instructions.

### Syntax

`.486`

### Remarks

The `.486` directive is not available in [M510](#) mode.

---

## `.486P` (Select 80486 Processor Protected Mode Instruction Set)

Enables assembly of all instructions (including privileged) for the 80486 processor. Also enables 80387 (and later) floating point instructions.

## Syntax

`.486P`

## Remarks

The **.486P** directive is not available in [M510](#) mode.

-----

# .586 (Select Pentium/586 Processor Instruction Set)

Enables assembly of instructions for the Pentium processor family. Also enables 80387 (and later) floating point instructions.

## Syntax

`.586`

## Remarks

The **.586** directive is not available in [M510](#) mode or [M600](#) mode.

-----

# .586P (Select Pentium/586 Processor Protected Mode Instruction Set)

Enables assembly of all instructions (including privileged) for the Pentium processor family. Also enables 80387 (and later) floating point instructions.

## Syntax

`.586P`

## Remarks

The **.586P** directive is not available in [M510](#) mode or [M600](#) mode.

-----

# .686 (Select Pentium Pro/686 Processor Instruction Set)

Enables assembly of instructions for the Pentium Pro processor family. Also enables 80387 (and later) floating point instructions.

## Syntax

`.686`

## Remarks

The **.686** directive is not available in the [M510](#), [M600](#), or [M611](#) emulation modes.

---

# .686P (Select Pentium Pro/686 Processor Protected Mode Instructions)

Enables assembly of all instructions (including privileged) for the Pentium Pro processor family. Also enables 80387 (and later) floating point instructions.

## Syntax

**.686P**

## Remarks

The **.686P** directive is not available in the [M510](#), [M600](#), or [M611](#) emulation modes.

---

# .MMX (Select MMX Processor Instruction Set Extensions)

Enables recognition of mnemonics for the MMX instruction set extensions.

## Syntax

**.MMX**

## Remarks

If 586 mnemonics (or later) are not already being recognized, the **.MMX** directive also causes an implicit **.586** directive to be executed.

Issuing any **.486** (or earlier) processor selection directive causes recognition of MMX mnemonics to be disabled.

If MMX mnemonics are being recognized, issuing a **.586** (or later) processor selection directive does not cause recognition of MMX mnemonics to be disabled. If this behavior is desired, use the **.NOMMX** directive.

The **.MMX** directive is not available in the [M510](#), [M600](#), or [M611](#) emulation modes.

---

# .NOMMX (Deselect MMX Processor Instruction Set Extensions)

Disables recognition of mnemonics for the MMX instruction set extensions.

## Syntax

**.NOMMX**

## Remarks

Does not affect recognition of instruction mnemonics for the currently selected primary processor; it only disables recognition of the MMX mnemonics.

The **.NOMMX** directive is not available in the [M510](#), [M600](#), or [M611](#) emulation modes.

## Example

```
; Top of file - no processor currently selected
.MMX           ; enables both MMX and 586 mnemonics
.NOMMX         ; 586 mnemonics still enabled
.686           ; 686 mnemonics now being recognized
.MMX           ; 686 and MMX mnemonics now being recognized
.NOMMX         ; 686 mnemonics still enabled
```

---

# .SIMD (Select SIMD Processor Instruction Set Extensions)

Enables recognition of mnemonics for the SIMD (single instruction, multiple data) instruction set extensions.

## Syntax

```
.SIMD
```

## Remarks

If 686 mnemonics (or later) are not already being recognized, the **.SIMD** directive also causes an implicit **.686** directive to be executed.

Issuing any **.586** (or earlier) processor selection directive causes recognition of SIMD mnemonics to be disabled.

If SIMD mnemonics are being recognized, issuing a **.686** (or later) processor selection directive does not cause recognition of SIMD mnemonics to be disabled. If this behavior is desired, use the **.NOSIMD** directive.

The **.SIMD** directive is not available in the [M510](#), [M600](#), [M611](#), [M612](#), or [M613](#) emulation modes.

---

# .NOSIMD (Deselect SIMD Processor Instruction Set Extensions)

Disables recognition of mnemonics for the SIMD (single instruction, multiple data) instruction set extensions.

## Syntax

```
.NOSIMD
```

## Remarks

Does not affect recognition of instruction mnemonics for the currently selected primary processor; it only disables recognition of the SIMD mnemonics.

The **.NOSIMD** directive is not available in the [M510](#), [M600](#), [M611](#), [M612](#), or [M613](#) emulation modes.

## Example

```
; Top of file - no processor currently selected
.586          ; 586 mnemonics now being recognized
.SIMD         ; enables both SIMD and 686 mnemonics
.NOSIMD       ; 686 mnemonics still enabled
```

---

## .XMM (Select SIMD Processor Instruction Set Extensions)

Enables recognition of mnemonics for the SIMD (single instruction, multiple data) instruction set extensions.

### Syntax

```
.XMM
```

### Remarks

This directive has the same functionality as the [.SIMD](#) directive; it is included for compatibility with MASM 6.14.

The **.XMM** directive is not available in the [M510](#), [M600](#), [M611](#), [M612](#), or [M613](#) emulation modes.

---

## Segments

A segment is a collection of instructions or data whose addresses are all relative to the same segment register. The code in your assembler language program defines and organizes segments.

You can define segments by using segment directives or full segment definitions.

This section describes the following directives used to create and manage segments:

```
ALIGN
.CODE
.CONST
.DATA
.DATA?
DOSSEG
.DOSSEG
ENDS
EVEN
.FARDATA
.FARDATA?
GROUP
.MODEL
ORG
SEGMENT
.SEQ
.STACK
```

---

## ALIGN (Align Code or Data Item)

Advances the current location counter to the next byte boundary that is a multiple of *Expression*.

## Syntax

```
ALIGN Expression
```

## Example

To align to a 2-byte boundary:

```
ALIGN 2
```

To align to a 4-byte boundary:

```
ALIGN 4
```

---

# .CODE (Opens Default or Named Code Segment)

Closes the currently opened segment (if any) and opens the default code segment or a segment with the name given by an optional *SegmentName* parameter. The **.CODE** directive may only be used if previous **.MODEL** directive has been processed.

## Syntax

```
.CODE [SegmentName]
```

## Remarks

When the *SegmentName* parameter is omitted from the **.CODE** directive, the assembler generates a default code segment whose name is determined by the memory model as follows:

Memory Model	Value for @code
TINY	_TEXT
SMALL	_TEXT
MEDIUM	<i>module</i> _TEXT
COMPACT	_TEXT
LARGE	<i>module</i> _TEXT
HUGE	<i>module</i> _TEXT
FLAT	CODE32

The *module* entry is replaced with base file name of the top-level module being assembled.

When operating in **M510** mode, the *SegmentName* parameter may only be specified for those memory models that allow multiple code segments (MEDIUM, LARGE, and HUGE), and the value of the **@code** symbol is not altered from the default. For other modes of operation, the *SegmentName* parameter is allowed for any model other than TINY, and the **@code** symbol is updated to reflect the *SegmentName* value.

---

# .CONST (Opens Default Constant Data Segment)

When used with **.MODEL**, starts a constant data segment for initialized read-only data.

## Syntax

`.CONST`

#### Remarks

The name of the segment is CONST32 in flat model, and CONST for all other models.

---

## `.DATA` (Opens Default Data Segment)

When used with **.MODEL**, starts a near data segment for initialized data.

#### Syntax

`.DATA`

#### Remarks

The name of the segment is DATA32 in flat model, and `_DATA` for all other models.

---

## `.DATA?` (Opens Default Uninitialized Data Segment)

When used with **.MODEL**, starts a near data segment for uninitialized data.

#### Syntax

`.DATA?`

#### Remarks

The name of the segment is BSS32 in flat model, and `_BSS` for all other models.

---

## `.DOSSEG/DOSSEG` (Specify Standard DOS Segment Ordering)

Orders the segments according to the DOS segment convention: CODE first, then segments not in DGROUP, and then segments in DGROUP. The segments in DGROUP follow this order:

1. Segments not in BSS or STACK
2. BSS segments
3. STACK segments

#### Syntax

`.DOSSEG` (preferred form)  
or

**Remarks**

**.DOSSEG** is the preferred form.

Use of this directive allows the linker to control the segment ordering according to conventions used in many high-level languages.

## ENDS (Close a Segment, Structure, or Union Declaration)

Closes a program segment opened with **SEGMENT** directive, or ends a structure or union definition opened with the **STRUCT** or **UNION** directives. Every **SEGMENT**, **STRUCT**, and **UNION** directive must end with a corresponding **ENDS** directive.

**Syntax**

```
Segment-Name ENDS
or
Structure-Name ENDS
or
Union-Name ENDS
```

**Remarks**

If the **ENDS** directive is not used with the corresponding **SEGMENT**, **STRUCT**, or **UNION** directive, an error occurs. An unmatched **ENDS** also causes an error.

**Note:** See the [SEGMENT](#), [STRUCT](#), and [UNION](#) directives for more details and examples of the use of **ENDS**.

**Example**

```
CONST    SEGMENT    word public 'CONST'
SEG1     DW          ARRAY_DATA
SEG2     DW          MESSAGE_DATA
CONST    ENDS
```

## EVEN (Align Code or Data Item on an Even Boundary)

The **EVEN** directive causes the program counter to go to an even boundary (an address that begins a word). This ensures that the code or data that follows is aligned on an even boundary.

**Syntax**

```
EVEN
```

**Remarks**

If the program counter is not already at an even boundary, **EVEN** causes the assembler to add a **NOP** (no operation) instruction so that the counter reaches an even boundary. An error message occurs if **EVEN** is used with a

byte-aligned segment. If the program counter is already at an even boundary, **EVEN** does nothing.

### Example

Before: PC points to 0019 hex (25 decimal).

EVEN

After: PC points to 001A hex (26 decimal).

-----

## .FARDATA (Opens Default or Named Far Data Segment)

When used with **.MODEL**, starts a far data segment for initialized data.

### Syntax

```
.FARDATA [SegmentName]
```

### Remarks

If the *SegmentName* parameter is not specified, the assembler sets it to FAR\_DATA.

-----

## .FARDATA? (Opens Default or Named Uninitialized Far Data Segm

When used with **.MODEL**, starts a far data segment for uninitialized data.

### Syntax

```
.FARDATA? [SegmentName]
```

### Remarks

If the *SegmentName* parameter is not specified, the assembler sets it to FAR\_BSS.

-----

## GROUP (Treat Multiple Segments as a Single Unit)

The **GROUP** directive associates a group *Name* with one or more segments, and causes all labels and variables defined in the given segments to have addresses relative to the beginning of the group, rather than to the segments where they are defined.

### Syntax

```
Name GROUP Segment-Name [, ...]
```

## Remarks

Each *Segment-Name* entry must be a unique segment name assigned by the **SEGMENT** directive. A *Segment-Name* entry may be a forward reference to a subsequently declared segment name.

An additional occurrence of a given group *Name* in a subsequent **GROUP** directive does not constitute a redefinition, but instead the effect is cumulative. The group *Name* itself is declared the first time it appears in a **GROUP** directive, but the group definition is not complete until the end of the source module is reached. The final group definition is the cumulative list of all unique segments named in all occurrences of a **GROUP** directive for that group *Name*.

Segments in a group need not be contiguous. Segments that do not belong to the group can be loaded between segments that do belong to the group. The only restriction is that for USE16 segments the distance (in bytes) between the first byte in the first segment of the group and the last byte in the last segment must not exceed 65535 bytes.

Group names can be used with the **ASSUME** directive and as an operand prefix with the segment override operation (:).

## Example

The following example shows how to use the **GROUP** directive to combine segments:

### In Module A:

```
CGROUP    GROUP    XXX,YYY
XXX        SEGMENT
           ASSUME   CS:CGROUP
           .
           .
           .
XXX        ENDS
YYY        SEGMENT
           ASSUME   CS:CGROUP
           .
           .
           .
YYY        ENDS
```

### In Module B:

```
CGROUP    GROUP    ZZZ
ZZZ        SEGMENT
           ASSUME   CS:CGROUP
           .
           .
           .
ZZZ        ENDS
```

The next example shows how to set **DS** with the paragraph number of the group called **DGROUP**.

### As immediate:

```
MOV        AX,DGROUP
MOV        DS,AX
```

### In assume:

```
ASSUME     DS:DGROUP
```

### As an operand prefix:

```
MOV        BX,OFFSET DGROUP:FOO
DW         FOO
DW         DGROUP:FOO
```

### Note:

1. **DW FOO** returns the offset of the symbol within its segment.

2. **DW DGROUP:FOO** returns the offset of the symbol within the group.

The next example shows how you can use the **GROUP** directive to create a **.COM** file type.

```
PAGE      ,132
TITLE     GRPCOM - Use GROUP to create a DOS .COM file
;Use the DOS EXE2BIN utility to convert GRPCOM.EXE to GRPCOM.COM.

CG        GROUP      CSEG,DSEG      ;ALL SEGS IN ONE GROUP
DISPLAY   MACRO      TEXT
LOCAL    MSG
DSEG      SEGMENT    BYTE PUBLIC 'DATA'
MSG       DB          TEXT,13,10,"$"
DSEG      ENDS
;;Macro produces partly in DSEG,
;;partly in CSEG
          MOV         AH,9
          MOV         DX,OFFSET CG: MSG
;;Note use of group name
;;in producing offset
          INT 21H
          ENDM
DSEG      SEGMENT    BYTE PUBLIC 'DATA'
;Insert local constants and work areas here
DSEG      ENDS
CSEG      SEGMENT    BYTE PUBLIC 'CODE'
          ASSUME      CS:CG,DS:CG,SS:CG,ES:CG ;SET BY LOADER
          ORG 100H ;Skip to end of the PSP
ENTPT     PROC        NEAR ;COM file entry at 0100H
          DISPLAY     "USING MORE THAN ONE SEGMENT"
          DISPLAY     "YET STILL OBEYING .COM RULES"
          RET         ;Near return to DOS
ENTPT     ENDP
CSEG      ENDS
          END         ENTPT
```

## .MODEL (Define Program Memory Segmentation Model)

The **.MODEL** directive establishes a predefined set of definitions, conventions, and modifications to various default operating behaviors of the assembler. These adjustments are designed to simplify certain programming tasks and to allow a more seamless integration with routines written in high level languages.

### Syntax

```
.MODEL Memory-Model [, Language-Type] [, OS-Type] [, Stack-Distance]
```

#### *Memory-Model:*

TINY  
SMALL  
COMPACT  
MEDIUM  
LARGE  
HUGE  
FLAT

#### *Language-Type:*

BASIC  
C  
FORTRAN  
OPTLINK  
PASCAL  
STDCALL  
SYSCALL

#### *OS-Type:*

OS\_DOS  
OS\_OS2

*Stack-Distance:*  
FARSTACK  
NEARSTACK

## Remarks

The **.MODEL** directive should be placed at the top of the file, after any [processor control](#) directives, but before any of the following simplified segmentation directives are encountered:

- [.CODE](#)
- [.CONST](#)
- [.DATA](#)
- [.DATA?](#)
- [.FARDATA](#)
- [.FARDATA?](#)
- [.STACK](#)

Each of these directives close any segment that is currently opened, then open a different segment whose name and attributes are determined by the *Memory-Model* argument.

## Memory-Model

The fundamental purpose of establishing a programming memory model is to define how the program will be organized within the constraints of the segmented processor architecture. It defines whether there are single or multiple default code and data segments, or whether the default code and data segments are merged into a single segment. The operating system upon which the program will run is a determining factor of which memory models can be used. The following table describes these relationships.

Memory Model	Default Code	Default Data	Merged?	Operating Systems
Tiny	Near	Near	Yes	DOS
Small	Near	Near	No	DOS, 16-Bit OS/2, Win16
Medium	Far	Near	No	DOS, 16-Bit OS/2, Win16
Compact	Near	Far	No	DOS, 16-Bit OS/2, Win16
Large	Far	Far	No	DOS, 16-Bit OS/2, Win16
Huge	Far	Far	No	DOS, 16-Bit OS/2, Win16
Flat	Near	Near	Yes	32-Bit OS/2, Win32

The assembler creates the default code and data segments, then automatically generates an [ASSUME CS:@code](#) and an [ASSUME DS:@data](#) statement to refer to them.

## Language-Type

Specifies the default naming convention for all public identifiers written that to the object file, and the method whereby parameters are passed to procedures (the *calling convention*). See the section on the [PROC](#) directive for a detailed explanation of the effects of the *Language-Type* setting.

## OS-Type

This parameter identifies the target operating system upon which the program will run, and is provided for compatibility with other assemblers. ALP ignores this parameter.

## Stack-Distance

The **NEARSTACK** parameter causes the assembler to assume that the stack segment and the default data segment are the same, and that the DS register is equal to the SS register. This is the default setting. The assembler performs an automatic [ASSUME SS:@data](#) statement when a near stack is used.

The **FARSTACK** parameter causes the assembler to assume that the stack is in a different physical segment from that of the default data, and that SS register is not equal to DS. This is typically the case for code in a 16-bit dynamic link library that must use the caller's stack. The assembler performs an automatic [ASSUME SS:STACK](#) when this keyword

is used.

---

## ORG (Adjust Segment Location Counter)

The **ORG** directive sets the location counter to the value of *Expression*. Subsequent instructions are generated beginning at this new location.

### Syntax

```
ORG Expression
```

### Remarks

The assembler must know all names used in *Expression* on pass 1, and the value must be either absolute or in the same segment as the location counter.

The numeric value of *Expression* must not be a quantity larger than that which is representable by an unsigned integer having the same word size as the current segment.

You can use the current address operator (\$) to refer to the current value of the location counter.

### Example

```
ORG    120H
ORG    $+2    ;SKIP NEXT 2 BYTES
```

To conditionally skip to the next 256-byte boundary:

```
CSEG    SEGMENT    PAGE
BEGIN    =          $
        .
        .
        .
        IF ($-BEGIN) MOD 256
;IF NOT ALREADY ON 256 BYTE BOUNDARY
        ORG ($-BEGIN)+256 - (($-BEGIN) MOD 256)
        ENDIF
```

---

## SEGMENT (Open a Program Information Segment)

Defines or reopens a segment called *Segment-Name* which will contain all subsequently emitted code or data.

### Syntax

```
Segment-Name SEGMENT [align] [combine] [use] ['class']
```

### Remarks

A segment definition may be followed by zero or more segment attributes, at most one from each of the following selections:

<i><b>align</b></i>	Instructs the linker to align the segment at the next <i>align</i> boundary. One of:	
	<b>BYTE</b>	The next 8-bit boundary.
	<b>DWORD</b>	The next 32-bit boundary.
	<b>PAGE</b>	The next 256-byte boundary (4096 under 32-bit OS/2).
	<b>PARA</b>	The next 16-byte boundary (default).
	<b>WORD</b>	The next 16-bit boundary.
<i><b>combine</b></i>	Controls how the linker will combine this segment with identically-named segments from other modules. One of:	
	<b>AT <i>address</i></b>	Locates the segment at the absolute paragraph given by <i>address</i> .
	<b>COMMON</b>	Unioned with segments from other modules.
	<b>PRIVATE</b>	Will not be combined with other segments (default).
	<b>PUBLIC</b>	Concatenated to segments from other modules.
	<b>STACK</b>	Concatenated to segments from other modules. At load time:
		SS=beginning of segment SS:(E)SP=end of segment
<i><b>use</b></i>	Word size of the segment.	
	<b>USE16</b>	The segment will have a 16-bit word size.
	<b>USE32</b>	The segment will have a 32-bit word size.
<i><b>'class'</b></i>	Instructs the linker to order segments according to the class name given by <i>'class.'</i> Segments will not be combined if their class names differ.	

## .SEQ (Specifies Sequential Segment Ordering)

Orders segments sequentially (the default order).

### Syntax

```
.SEQ
```

## .STACK (Defines Default Stack Segment With Optional Size)

When used with **.MODEL**, defines a stack segment with the segment name STACK. The optional *Size* specifies the number of bytes for the stack (default 1024).

### Syntax

```
.STACK [Size]
```

### Remarks

The **.STACK** directive does not leave the stack segment open when the statement is completed, since it is not a common practice to emit initialized data into the stack segment.

The name of the segment is STACK32 in flat model, and STACK for all other models.

---

# Type Definition

Type definition directives allow the creation of user-defined data types.

This section describes the following type definition directives:

```
RECORD
STRUCT/STRUC
TYPDEF
UNION
```

---

## RECORD (Define a Record Type Name)

A record is a bit pattern you define to format bytes, words, or dwords for bit-packing. The *RecordName* becomes a *Record-TypeName* that can be used create record variables.

### Syntax

```
RecordName RECORD FieldDeclaration [, [LineBreak] FieldDeclaration ...]
```

Where *FieldDeclaration* has the following form:

```
FieldName:Width [ = InitialValue]
```

The optional *LineBreak* entry allows you to end a *FieldDeclaration* with a comma, enter an optional *EndOfLine-Comment* followed by a physical *NewLine* character, then continue the record definition on the next line.

### Remarks

The *RecordName* and *FieldName* entries are unique globally-scoped *Identifier*s that must be specified. Upon successful processing of the **RECORD** definition, the *RecordName* entry is converted to a *Record-TypeName*, and all *FieldNames* are converted to *Record-FieldNames*.

Each *Width* entry in a *FieldDeclaration* is specified as an *Expression* which must evaluate to an *Absolute-ExpressionType*. The cumulative value of all *Width* entries becomes the total *RecordWidth* and must not exceed 32, the size of a DWORD, the maximum size for a *Record-TypeName*. The *Operand Size* of the record becomes 1 (BYTE) if the *RecordWidth* is from 1 through 8, 2 (WORD) if the *RecordWidth* is from 9 through 16, and 4 (DWORD) if the *RecordWidth* is from 17 through 32. Any other value causes an error. If the total number of bits in the *RecordWidth* is not evenly divisible by the *Operand Size*, the assembler right-justifies the fields into the least-significant bit positions of the record.

When a *Record-FieldName* is referenced in an expression, the value returned is the shift value required to access the field. The **WIDTH** operator is used on the *Record-FieldName* to return the width of the field in bits, and the **MASK** operator is used to obtain the value necessary for isolating the field within the record.

The *InitialValue* entry contains the default value to used for the field when a record variable is allocated. If the field is at least 7 bits wide, you can initialize it to an ASCII character (for example, FIELD:7='Z').

To initialize a record, use the form:

```
[Identifier] Record-TypeName Expression [, Expression ...]
```

The *Identifier* entry is an optional name for the first byte, word, or dword of the reserved memory. The *Record-TypeName* defines the format and default field values, and is the *RecordName* you assigned to the record in the **RECORD** directive.

At least one *Expression* entry must be specified; additional entries are optional. The *Expression* must resolve to a *Compound-ExpressionType*, which may also be duplicated by specifying it as a sub-expression of a *Duplicated-ExpressionType*. Each *Compound-ExpressionType* represents a single allocated record entry; each explicit sub-expression of the *Compound-ExpressionType* represents a field value which overrides the default *InitialValue* for the field given in the record definition.

## Example

Define the record fields; begin with the most significant fields first:

```
MODULE RECORD R:7,      ; First field.  ", LineBreak" syntax
                  E:4,      ; may be used to split RECORD
                  D:5      ; definition across multiple lines
```

Fields are 7 bits, 4 bits, and 5 bits; the assembler gives no default value. Most significant byte first, this looks like:

```
RRRR RRRE EEED DDDD
```

To reserve its memory:

```
STG_FLD  MODULE  <7,,2> ; Initializer is a Compound-ExpressionType
```

This defines R=7 and D=2 and leaves E unknown; it produces 2 bytes, the least significant byte first:

```
02  0E
```

Define the record fields:

```
AREA  RECORD  FLA:8='A',FLB:8='B'
```

To reserve its memory:

```
CHAR_FLD  AREA <,'P'>
```

This defines FLA='A' (the default) and changes FLB='P'.

To use a field in the record:

```
DEFFIELD  RECORD  X:3,Y:4,Z:9
          .
          .
          .
TABLE     DEFFIELD 10 DUP(<0,2,255>)
          .
          .
          .
          MOV DX,TABLE[2]
; Move data from record to register
; other than segment register
          AND DX,MASK Y
; Mask out fields X and Y
; to remove unwanted fields
; The MASK of Y equals 1E00H
; 00011111000000000B (1E00H) Is the value
          MOV CL,Y          ; Get shift count
                          ; 9 is the value
          SHR DX,CL         ; Field to low-order
                          ; bits of register, DX is now
                          ; equal to the value of field Y
          MOV CL,WIDTH Y    ; Get number of bits
                          ; in field - 4 is the value,
                          ; the WIDTH of Y is 4
```

---

# STRUCT/STRUC (Define a Structure Type Name)

Defines a *Structure-TypeName* that represents an [aggregate](#) data type containing one or more fields.

## Syntax

```
Structure-Name STRUCT
    FieldDeclaration
    .
    .
    .
Structure-Name ENDS
```

Where *FieldDeclaration* has the following form:

```
[FieldName] Allocation-TypeName InitialValue [, InitialValue ...]
```

## Remarks

The obsolete spelling for the **STRUCT** directive is **STRUC**.

The syntax for the *FieldDeclaration* is that of a normal data allocation statement. See the section on [Data Allocation](#) for a full description of this syntax.

The various parts of the *FieldDeclaration* are described as follows:

### *FieldName*

Each *FieldName* entry is converted to *Structure-FieldName* when processing of the structure definition is complete. If this field is omitted and the *Allocation-TypeName* resolves to a *Structure-TypeName* or *Union-TypeName*, then all of the fields defined within the imbedded structure or union are *promoted* to be visible at the same level as other *FieldName* entries in the current structure given by the *Structure-Name*.

### *Allocation-TypeName*

The allowable values for this field are described in detail in the [Data Allocation](#) section. In modes other than [M510](#), the assembler accepts imbedded occurrences of other structures or unions by specifying an identifier that resolves to a *Structure-TypeName* or *Union-TypeName* in this field.

### *InitialValue*

The *InitialValue* field must be an *Expression* that resolves to an *ExpressionType* appropriate for the *Allocation-TypeName* utilized in the *FieldDeclaration*. The *InitialValue* expressions become part of the structure type definition. These values are used as default initializers when an instance of the structure is allocated and no explicit override values are specified for a particular field.

## Example

Define a *Structure-TypeName* called **Numbers**:

```
Numbers    STRUCT
    One     DB      0
    Two     WORD    0
            BYTE    3
    Four    DWORD   ?
Numbers    ENDS
```

Allocate a structure variable called **Values** using the **Numbers** *Structure-TypeName*, overriding the **One**, **Two**, and **Four** *Structure-FieldName* entries with explicit values, and the third (unnamed) entry is initialized with the default *InitialValue* inherited from the *FieldDeclaration*:

```
Values Numbers <1, 2, , 4>
```

-----

## TYPDEF (Create a User-Defined Type Name)

Defines a *Typedef-TypeName* that is an alias for another type declaration.

### Syntax

```
TypeName TYPDEF Type-Declaration
```

### Remarks

The *TypeName* entry is a unique globally-scoped *Identifier* that must be specified. Upon successful processing of the **TYPDEF** directive, the *TypeName* entry is converted to a *Typedef-TypeName* which can then be used in expressions or as a directive in data allocation statements.

The **TYPDEF** directive can be used to create a direct alias for another intrinsic type (a *Scalar-TypeName*, *Record-TypeName*, *Structure-TypeName*, *Union-TypeName*, or other *Typedef-TypeName*), a pointer to another type, or it can be used to create vector types (arrays).

### Examples

The following are examples of **TYPDEF** usage:

```
CHAR      typedef byte                ; CHAR is an alias for intrinsic type
PCHAR     typedef ptr CHAR            ; PCHAR is a pointer to CHAR

BUFFER_T  struct
  pLetter PCHAR  ?                    ; current position in buffer
  Letters CHAR   "ABCDEF",0           ; array of characters
BUFFER_T  ends

BUFFER    typedef BUFFER_T            ; alias for intrinsic type
PBUFFER   typedef ptr BUFFER_T        ; pointer to the BUFFER type

DATA      SEGMENT
HexChars  BUFFER  <>                  ; allocate structure via typedef
pHexChars PBUFFER offset HexChars     ; point to the allocated structure
DATA      ENDS
```

-----

## UNION (Define a Union Type Name)

Defines a *Union-TypeName* that represents an *aggregate* data type containing one or more fields. All of the fields occupy the same physical position in storage.

### Syntax

```

Union-Name UNION
    FieldDeclaration
    .
    .
    .
Union-Name ENDS

```

Where *FieldDeclaration* has the following form:

```
[FieldName] Allocation-TypeName InitialValue [, InitialValue ...]
```

## Remarks

This directive is not available in [M510](#) mode.

The syntax for the *FieldDeclaration* is that of a normal data allocation statement. See the section on [Data Allocation](#) for a full description of this syntax.

The various parts of the *FieldDeclaration* are described as follows:

### *FieldName*

Each *FieldName* entry is converted to [Union-FieldName](#) when processing of the union definition is complete. If this field is omitted and the [Allocation-TypeName](#) resolves to a [Structure-TypeName](#) or [Union-TypeName](#), then all of the fields defined within the imbedded structure or union are **promoted** to be visible at the same level as other *FieldName* entries in the current union given by the *Union-Name*.

### [Allocation-TypeName](#)

The allowable values for this field are described in detail in the [Data Allocation](#) section. The assembler accepts imbedded occurrences of other structures or unions by specifying an identifier that resolves to a [Structure-TypeName](#) or [Union-TypeName](#) in this field.

### *InitialValue*

The *InitialValue* field must be an [Expression](#) that resolves to an [ExpressionType](#) appropriate for the [Allocation-TypeName](#) utilized in the *FieldDeclaration*. Only the *InitialValue* expression for the first field becomes part of the union type definition; expressions specified for the remaining fields are ignored. This value is used as the default initializer when an instance of the union is allocated and no explicit override value is specified for the field.

## Example

```

        .386
IS_sint32 equ -4
IS_sint16 equ -2
IS_sint8  equ -1
NO_TYPE  equ  0
IS_uint8  equ  1
IS_uint16 equ  2
IS_uint32 equ  4

TYPE_T    typedef  SBYTE

DATA_T    union
    uint8  BYTE    ?
    sint8  SBYTE    ?
    uint16 WORD     ?
    sint16 SWORD    ?
    uint32 DWORD    ?
    sint32 SDWORD   ?
DATA_T    ends

VALUE_T    struct
    DataType  TYPE_T NO_TYPE
    DataValue DATA_T {}
VALUE_T    ends

```

```

        .data
Value    VALUE_T { IS_uint8, { 1 } }          ; unsigned 8-bit value of 1

        .code

; Procedure: IsNegative
; Returns  : 1 in EAX if Value.DataValue holds a negative number
;           0 in EAX if Value.DataValue holds a positive number

IsNegative proc
        cmp     Value.DataType, NO_TYPE       ; check sign of TYPE_T
        jns     short Positive                ; if positive, so is value

; check for signed 8-bit integer
        cmp     Value.DataType, IS_sint8
        jne     short @F                      ; not 8, check for 16
        movsx   EAX, Value.DataValue.sint8    ; convert 8 bits to 32
        jmp     short Check                   ; and check the value

; check for signed 16-bit integer
@@:     cmp     Value.DataType, IS_sint16
        jne     short @F                      ; not 16, check for 32
        movsx   EAX, Value.DataValue.sint16   ; convert 16 bits to 32
        jmp     short Check                   ; and check the value

; check for signed 32-bit integer
@@:     cmp     Value.DataType, IS_sint32
        jne     short Positive                ; unknown, assume positive
        mov     EAX, Value.DataValue.sint32   ; get full 32 bit number

Check:  or      EAX, EAX                      ; check for negative value
        jns     short Positive                ; no sign bit, positive
        mov     EAX, 1                       ; indicate negative
        ret                                     ; and return

Positive: mov    EAX, 0                       ; indicate positive
        ret                                     ; and return

IsNegative endp
end

```

## Miscellaneous

This section describes the following miscellaneous directives:

```

=
.ABORT
ASSUME
EQU
LABEL
OPTION
.RADIX

```

## = (Assign an Expression to an Assembler Variable)

The = directive lets you create a symbolic assembler-time variable. Numeric expressions may be assigned to the variable as many times as necessary.

### Syntax

*Name* = *Expression*

## Remarks

The `=` directive is similar to the `EQU` assembler directive except you can redefine *Name* without causing an error condition. However, the `=` directive is more restrictive about the allowable *ExpressionType*s that can be utilized in the *Expression* field, and it cannot be used to create *Text-EquateName*s.

*Name* is a globally-scoped *Identifier*. The *Expression* entry must evaluate to an *Operand-ExpressionType*. If an evaluation error occurs, or if the *Expression* references an external identifier, or if the *Expression* evaluates to one of the following *Operand-ExpressionType*s:

- *Indexed-ExpressionType*
- *Register-ExpressionType*
- *Floating-Point-ExpressionType*
- *Compound-ExpressionType*
- *Duplicated-ExpressionType*

then an error message is issued and the assignment does not take place. Otherwise, the *Identifier* is converted to a *Numeric-EquateName*.

See also the `EQU` assembler directive and the `EQU` preprocessor directive.

## Example

```
EMP = 6           ;Establish as redefineable numeric equate
EMP EQU 6         ;OK, value is the same, EMP remains redefineable
EMP EQU 7         ;Error, can't change value with EQU
EMP = 7           ;OK, EMP is redefineable with =
EMP = EMP+1       ;Can refer to its previous definition
```

**Note:** The *Expression-Attribute*s inherited from the *Expression* during an assignment are not retained in subsequent assignments. For example:

```
VECTOR = WORD PTR 4      ; Type-Declaration attribute
      MOV [BX],VECTOR     ; Store the 4 as a word
VECTOR = 6               ; Type-Declaration attribute discarded
      MOV [BX],VECTOR     ; Error, no size for operands
```

---

# .ABORT (Terminate the Assembly)

Terminates the assembly at the point where the `.ABORT` directive is encountered. The remainder of the input stream is not read.

## Syntax

```
.ABORT
```

## Remarks

The `.ABORT` directive is only available in `ALP` mode

---

# ASSUME (Inform Assembler of Register Contents)

The `ASSUME` directive establishes an assembly-time association between a machine register and a program object or data type. By

informing the assembler of the type of information to which a register points, certain programming tasks can be simplified and the assembler can perform some operations automatically.

## Syntax

```
ASSUME Association [, Association ...]
```

### *Association:*

*Segment-Register-Association*  
*General-Purpose-Register-Association*  
**NOTHING**

### *Segment-Register-Association:*

*Segment-Register* : *Expression*  
*Segment-Register* : **NOTHING**

### *General-Purpose-Register-Association:*

*General-Purpose-Register* : *Type-Declaration*  
*General-Purpose-Register* : **NOTHING**

## Remarks

If the **NOTHING** keyword is specified for the *Association* field, all register associations are cancelled.

If the **NOTHING** keyword is specified for a particular *Segment-Register* or *General-Purpose-Register*, only the association for that register is cancelled.

The following sections describe the two types of register associations:

- *Segment-Register-Association*
- *General-Purpose-Register-Association*

---

# Segment Register Association

A *Segment-Register-Association* establishes an assembly-time association between a *Segment-Register* and an expression that resolves to a *GroupName* or *SegmentName*. It allows the programmer to describe for the assembler what values are held in the segment registers at program run-time.

When the user program executes, all instructions that access memory do so through a particular segment register. To generate the correct encoding for an instruction that accesses a memory location, the assembler must know which segment register will be used in the effective memory address. In general, accessing a memory location from within a user program is done by referencing a named variable defined within a particular named segment.

Before accessing a named program variable (in a named memory segment), it is the programmer's responsibility to insure that the desired segment register actually references the correct physical segment at program run-time. Unless the **ASSUME** directive is used to describe this association, the assembler has no way of knowing *which* segment register (if any) is addressing a named segment when a reference to a named variable contained therein is encountered. In this situation, the programmer is forced to use the *Segment Override (: Operator)* in every instruction to "reach" the desired variable and cause the assembler to generate the proper instruction encoding. The association established by the **ASSUME** directive allows the assembler to take over the task of verifying memory references and generating the appropriate instructions.

If you temporarily use a segment register to contain a value other than the segment or group identified in the **ASSUME** association, then you should reflect the change with a new **ASSUME** statement, or cancel the association with an **ASSUME xS:NOTHING** construct.

When the contents of a segment register are used for addressability, the register value should never contradict the association established for that register.

When the **.MODEL** directive is utilized and the program is designed to follow the conventions that it establishes, the **ASSUME** directive is no longer needed in most cases.

## Example

```

Data SEGMENT
Stuff WORD 0
Data ENDS

Code SEGMENT
ASSUME NOTHING ; Cancel all register assumptions
mov ax,Data ; Load general-purpose register with segment frame,
mov DS,ax ; then establish addressability through DS
mov ES,ax ; and ES. The assembler doesn't "know" this yet
mov Stuff, 1 ; Error, can't reach Stuff
ASSUME ES:Data ; Associate ES register with Data segment
add Stuff, bx ; Now we can reach Stuff, but the assembler needs
; to generate an ES override instruction byte
ASSUME DS:SEG Stuff ; Expression to extract the segment value of Stuff
; This has the same effect as ASSUME DS:Data
; Now both DS and ES are associated with Data
add Stuff, cx ; This time, the instruction doesn't need an
; override byte because DS is the default
; register for normal accesses to memory
ASSUME DS:NOTHING ; Cancel the association between DS and Data
add Stuff, dx ; Once again, the ES override is generated
add DS:Stuff, dx ; Must use "force" if we want the default encoding
Code ends
end

```

### Warning:

If an **ASSUME CS:***Expression* is placed before the code segment it is referencing, the assembler will ignore the **ASSUME**. The **ASSUME CS:***Expression* statement must follow the **SEGMENT** definition statement of the code segment it is referencing.

The **ASSUME** statement for the **CS** register should be placed immediately following the code **SEGMENT** statement, before any labels are defined in that code segment.

## General-Purpose Register Association

A **General-Purpose-Register-Association** establishes an assembly-time association between a *General-Purpose-Register* and a *Type-Declaration*. It allows the programmer to describe for the assembler what type of data is being held in the general purpose register at program run-time.

This feature can be very useful when the programmer is treating a general-purpose register as a "pointer" to a particular type of storage. If this "pointer" is being utilized many times in the program, (perhaps changing in value but never in the type of data to which it points), the **ASSUME** directive can be used to associate the register with the type of data to which it points. This frees the programmer from having to use an explicit *Type Conversion (PTR Operator)* every time the register is used to access memory.

A register may only be associated with a data type whose operand size matches that of the register. For instance, the following construct is illegal:

```
ASSUME EBX:BYTE ; Error, EBX is a DWORD register
```

The most useful situation is for the register to contain a pointer to another data type. In this situation, the *Indirection ([] Operator)* may be used store or retrieve data through the register without the need for an explicit conversion operation:

```

ASSUME EDI:NOTHING ; This is the assembler default setting
MOV [EDI], 1 ; What is the size supposed to be?
MOV byte ptr [EDI], 1 ; Fixes the problem, but this can get tiring
ASSUME EDI:PTR BYTE ; EDI is now a pointer to a byte
MOV [EDI], 1 ; assembler knows what to do with this now
INC [EDI] ; and this too

```

The following constructs are legal but not particularly useful since they simply restate what is already known about the registers (the operand size), and the assembler doesn't enforce a strict level of type checking against register operands:

```

ASSUME ECX:SDWORD ; Signed double-word matches size of ECX
ASSUME EBX:DWORD ; Unsigned double-word matches size of EBX
MOV ECX, 0FFFFFFEh ; Register type-checking is not strict
MOV EBX, -1 ; enough to flag these as errors

```

In fact, any data type that matches the size of the register may be used; the assembler checks the sizes and reports mismatches, but effectively ignores any settings that are not pointers to other types. Consider the following example:

```
STRUCT_T STRUCT
    One    BYTE 1
    Two    BYTE 2
    Three  BYTE 3
    Four   BYTE 4
STRUCT_T ENDS

ASSUME EBX:STRUCT_T      ; Ok, STRUCT_T is 4 bytes in size
MOV     EBX, -1          ; Legal, but not very meaningful...

; A more useful situation (given that EBX is now holding data of type
; STRUCT_T) would be for the assembler to allow the following notation:

MOV     EBX, { 4, 3, 2, 1 } ; Hypothetical (UNSUPPORTED!) syntax...

; It would also be nice at this point if the symbolic debugger could
; show us the value of EBX in the appropriate format, but the assembler
; does not support the emitting of context-sensitive symbolic debugging
; information.
```

## EQU (Assign an Expression to a Symbolic Constant)

The **EQU** directive assigns the value of *Expression* to *Name*.

### Syntax

```
Name EQU Expression
```

### Remarks

If *Name* has already been defined as a *Numeric-EquateName* and its currently assigned value differs from the value given by *Expression*, an error message is produced. Unlike symbols created with the = (equal sign) directive, symbols created with the **EQU** directive cannot be redefined with different values.

The *Expression* entry must evaluate to an *Operand-ExpressionType*. If an evaluation error occurs or if the *Expression* evaluates to one of the following *Operand-ExpressionType*s:

- *Indexed-ExpressionType*
- *Floating-Point-ExpressionType*
- *Compound-ExpressionType*
- *Duplicated-ExpressionType*

then the *Identifier* is converted to a *Text-EquateName*. Otherwise, the *Identifier* is converted to a *Numeric-EquateName*.

See also **EQU** and **=**.

### Example

```
A    EQU    <BP +>    ;explicit text literal, A is a text equate
B    EQU    BP +      ;invalid expression - text equate equivalent to A
B    EQU    1 + 2      ;valid expression - but still a text equate <1 + 2>
C    EQU    1 + 2      ;converted to assembler symbolic constant, value = 3
C    EQU    <3>        ;illegal, cannot convert back to text equate
```

---

# LABEL (Associate a Symbolic Name With Current Address)

The **LABEL** directive defines the following attributes of *Name*:

- Segment: current segment being assembled
- Offset: current position within this segment
- Type: the operand of the **LABEL** directive

## Syntax

```
Name LABEL Type-Declaration
      or
Name :
      or
Name ::
```

## Remarks

The **LABEL** directive provides a method of labeling a memory location and assigning it a type without allocating any storage. It can be used to create multiple labels of differing types that are aliases for the same memory location.

The *Name* entry is an *Identifier* that is converted to a *LabelName* according to the value given by *Type-Declaration*. See the section on [label names](#) for more information on the details of this conversion.

The : and :: forms of this directive are used for defining code labels. In this case, the *Name* entry is converted to a *Target-LabelName*. The double-colon form of the directive is used when the *Name* must be visible outside of the procedure block in which it is defined.

## Example

To refer to a data area but use a length different from the original definition of that area:

```
BARRAY LABEL BYTE
ARRAY DW 100 DUP(0)
      .
      .
      .
      ADD AL,BARRAY[99] ;ADD 100th BYTE TO AL
      ADD AX,ARRAY[98] ;ADD 50th WORD TO AX
```

To define multiple entry points within a procedure:

```
SUBRT PROC FAR
      .
      .
      .
SUB2 LABEL FAR ;Should have same attribute as containing PROC
      .
      .
      .
      RET
SUBRT ENDP
```

---

# OPTION (Modify Default Behaviors)

The **OPTION** directive allows the user to alter certain default behaviors of the assembler, normally to provide backward compatibility with older assemblers. The **OPTION** directive is not available when assembling in [M510](#) mode.

## Syntax

```
OPTION Option-Item [, [LineBreak] Option-Item ...]
```

## Remarks

The *Option-Item* arguments are defined as follows (the underlined keywords denote the default values):

**DOTNAME** | **NODOTNAME**

Allows user identifiers to begin with an introductory dot (.) character.

**EXPR16** | **EXPR32**

Specifies whether expressions are evaluated using 16-bit or 32-bit arithmetic. Some programs may require reverting back to **EXPR16** in order to assemble without problems. Once this value has been set it cannot be changed. The use of a processor selection directive to select a 32-bit processor is equivalent to selecting **OPTION EXPR32**, which prevents any further attempt to select **OPTION EXPR16**.

**LANGUAGE:** *Language-Name*

Specifies the default language type for identifiers with **PUBLIC** or **EXPORT** visibility. This option overrides any setting given in the [.MODEL](#) directive.

**OFFSET:** *Offset-Type*

Determines how relocatable offset values are written to the object file

**OLDSTRUCTS** | **NOOLDSTRUCTS**

**PROC:** *Visibility*

**SCOPED** | **NOSCOPE**D

**SEGMENT:** *Address-Size*

output,  
encoded in the  
form of a linker  
"fixup" record.  
The possible  
values for  
*Offset-Type*  
are  
**SEGMENT**,  
**GROUP**, and  
**FLAT**.

The  
**OLDSTRUCT**  
**S** keyword  
causes  
structure field  
names to  
become global  
identifiers  
rather than  
local names  
private to the  
structure type.  
It also  
prevents the  
[Structure/Union Field Selection \(. Operator\)](#) from  
performing  
strict checking  
on its  
operands,  
requiring its  
left operand to  
have a  
structure type  
and its right  
operand to be  
the name of a  
field contained  
therein.

Specifies the  
default visibility  
for procedure  
names. This  
can be one of  
**PRIVATE**,  
**PUBLIC**, or  
**EXPORT**.

The  
**NOSCOPE**D  
keyword forces  
all code label  
names defined  
within  
procedures to  
be visible to  
the entire  
module and  
not just from  
within the  
defining  
procedure.

Explicitly sets  
the default  
address size  
value. This is  
used to control  
the address  
size of  
segments that

are opened without explicit **USE16** or **USE32** keywords, and of global identifiers that are declared outside of segment boundaries. The possible values for *Address-Size* are **USE16**, **USE32**, and **FLAT**.

---

## .RADIX (Set the Default Base for Numeric Literals)

The **.RADIX** directive lets you change the default **RADIX** (decimal) to base 2, 8, 10, or 16.

### Syntax

```
.RADIX Expression
```

### Remarks

The *Expression* entry is in decimal radix regardless of the current radix setting.

The **.RADIX** directive does not affect real numbers initialized as variables with **DD**, **DQ**, or **DT**.

When using **.RADIX 16**, be aware that if the hex constant ends in either B or D, the assembler thinks that the B or D is a request to cancel the current radix specification with a base of binary or decimal, respectively. In such cases, add the H base override (just as if **.RADIX 16** were not in use).

### Example

The statement:

```
.RADIX 16  
DW 120B
```

produces an error because 2 is not a valid binary number. The correct specification is:

```
DW 120BH
```

The following example:

```
.RADIX 16  
DW 89CD
```

also produces an error because C is not a valid decimal number. The correct specification is:

```
DW 89CDH
```

The dangerous case is when no error is produced. For example:

```
.RADIX 16
DW 120D
```

produces a constant whose value is 120 decimal, not '120D' hex, which might have been the intended value.

The following two move instructions are the same:

```
MOV BX,OFFH
.RADIX 16
MOV BX,OFF
```

The following example:

```
.RADIX 8
DQ 19.0 ; Treated as decimal
```

produces a constant whose value is 19 decimal because 19.0 is a real number. However, if you leave off the decimal point, the following:

```
.RADIX 8
DQ 19 ; uses current radix
```

produces a syntax error because nine is not a valid number in .RADIX 8.

---

## Processor Reference

This chapter presents an overview of the instruction set and lists the complete instruction set in alphabetical order. For each instruction, the forms are given for each operand combination, including object code produced, operands required, execution time, and a description. For each instruction, there is an operational description and a summary of exceptions generated.

---

## Intel Instruction Set Overview

This section contains an introduction to the Intel instruction set and presents the terminology necessary to understand the encoding and operation of each individual instruction.

---

## Operand-Size and Address-Size Attributes

When executing an instruction, the processor can address memory using either 16 or 32-bit addresses. Consequently, each instruction that uses memory addresses has associated with it an address-size attribute of either 16 or 32 bits. The use of 16-bit addresses implies both the use of 16-bit displacements in instructions and the generation of 16-bit address offsets (segment relative addresses) as the result of the effective address calculations. 32-bit addresses imply the use of 32-bit displacements and the generation of 32-bit address offsets. Similarly, an instruction that accesses words (16 bits) or doublewords (32 bits) has an operand-size attribute of either 16 or 32 bits.

The attributes are determined by a combination of defaults, instruction prefixes, and (for programs executing in protected mode) size-specification bits in segment descriptors.

---

## Default Segment Attribute

For programs running in protected mode, the D bit in executable-segment descriptors specifies the default attribute for both address size and operand size. These default attributes apply to the execution of all instructions in the segment. A clear D bit sets the default address size and operand size to 16 bits; a set D bit, to 32 bits.

Programs that execute in real mode or virtual-8086 mode have 16-bit addresses and operands by default.

## Operand-Size and Address-Size Instruction Prefixes

The internal encoding of an instruction can include two byte-long prefixes: the address-size prefix, 67H, and the operand-size prefix, 66H. (A later section, "Instruction Format," shows the position of the prefixes in an instruction's encoding.) These prefixes *override* the default segment attributes for the instruction that follows. The following table shows the effect of each possible combination of defaults and overrides.

### Effective Size Attributes

Segment Default D = ...	0	0	0	0	1	1	1	1
Operand-Size Prefix 66H	N	N	Y	Y	N	N	Y	Y
Address-Size Prefix 67H	N	Y	N	Y	N	Y	N	Y
Effective Operand Size	16	16	32	32	32	32	16	16
Effective Address Size	16	32	16	32	32	16	32	16

Y = Yes, this instruction prefix is present  
N = No, this instruction prefix is not present

## Address-Size Attribute for Stack

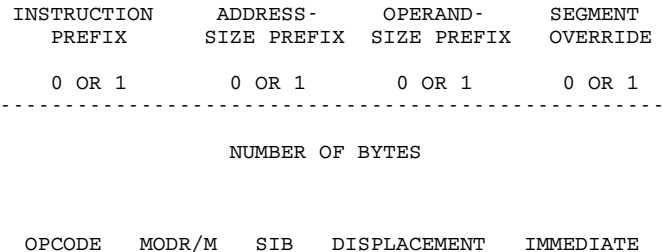
Instructions that use the stack implicitly (for example: POP EAX) also have a stack address-size attribute of either 16 or 32 bits. Instructions with a stack address-size attribute of 16 use the 16-bit SP stack pointer register; instructions with a stack address-size attribute of 32 bits use the 32-bit ESP register to form the address of the top of the stack.

The stack address-size attribute is controlled by the B bit of the data-segment descriptor in the SS register. A value of zero in the B bit selects a stack address-size attribute of 16; a value of one selects a stack address-size attribute of 32.

## Instruction Format

All instruction encodings are subsets of the general instruction format shown in the following figure. Instructions consist of optional instruction prefixes (in any order), one or two primary opcode bytes, possibly an address specifier consisting of the ModR/M byte and the SIB (Scale Index Base) byte, a displacement, if required, and an immediate data field, if required.

### Instruction Format



1 OR 2	0 OR 1	0 OR 1	0,1,2 OR 4	0,1,2 OR 4
-----				
NUMBER OF BYTES				

Smaller encoding fields can be defined within the primary opcode or opcodes. These fields define the direction of the operation, the size of the displacements, the register encoding, or sign extension; encoding fields vary depending on the class of operation.

Most instructions that can refer to an operand in memory have an addressing form byte following the primary opcode byte(s). This byte, called the ModR/M byte, specifies the address form to be used. Certain encodings of the ModR/M byte indicate a second addressing byte, the SIB (Scale Index Base) byte, which follows the ModR/M byte and is required to fully specify the addressing form.

Addressing forms can include a displacement immediately following either the ModR/M or SIB byte. If a displacement is present, it can be 8-, 16- or 32-bits.

If the instruction specifies an immediate operand, the immediate operand always follows any displacement bytes. The immediate operand, if specified, is always the last field of the instruction.

Zero or one bytes are reserved for each group of prefixes. The prefixes are grouped as follows:

1. Instruction Prefixes: REP, REPE/REPZ, REPNE/REPNZ, LOCK
2. Segment Override Prefixes: CS, SS, DS, ES, FS, GS
3. Operand Size Override
4. Address Size Override

For each instruction, one prefix may be used from each group. The effect of redundant prefixes (more than one prefix from a group) is undefined and may vary from processor to processor. The prefixes may come in any order.

The following are the allowable instruction prefix codes:

F3H	REP prefix (used only with string instructions)
F3H	REPE/REPZ prefix (used only with string instructions)
F2H	REPNE/REPZ prefix (used only with string instructions)
F0H	LOCK prefix

The following are the segment override prefixes:

2EH	CS segment override prefix
36H	SS segment override prefix
3EH	DS segment override prefix
26H	ES segment override prefix
64H	FS segment override prefix
65H	GS segment override prefix
66H	Operand-size override
67H	Address-size override

-----

## ModR/M and SIB Bytes

The ModR/M and SIB bytes follow the opcode byte(s) in many of the processor instructions. They contain the following information:

- The indexing type or register number to be used in the instruction
- The register to be used, or more information to select the instruction
- The base, index, and scale information

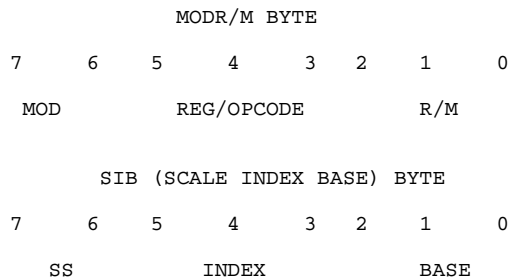
The ModR/M byte contains three fields of information:

- The **mod** field, which occupies the two most significant bits of the byte, combines with the *r/m* field to form 32 possible values: eight registers and 24 indexing modes.
- The **reg** field, which occupies the next three bits following the mod field, specifies either a register number or three more bits of opcode information. The meaning of the reg field is determined by the first (opcode) byte of the instruction.
- The ***r/m*** field, which occupies the three least significant bits of the byte, can specify a register as the location of an operand, or can form part of the addressing-mode encoding in combination with the **mod** field as described above.

The based indexed forms of 32-bit addressing require the SIB byte. The presence of the SIB byte is indicated by certain encodings of the ModR/M byte. The SIB byte then includes the following fields:

- The **ss** field, which occupies the two most significant bits of the byte, specifies the scale factor.
- The **index** field, which occupies the next three bits following the **ss** field and specifies the register number of the index register.
- The **base** field, which occupies the three least significant bits of the byte, specifies the register number of the base register.

#### ModR/M and SIB Byte Formats



## 16-Bit Addressing Forms with the ModR/M Byte

r8 (/r)	AL	CL	DL	BL	AH	CH	DH	BH	
r16 (/r)	AX	CX	DX	BX	SP	BP	SI	DI	
r32 (/r)	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI	
/digit (Opcode)	0	1	2	3	4	5	6	7	
REG =	000	001	010	011	100	101	110	111	

Effective Address	MOD	R/M	MODR/M Values in Hexadecimal							
[BX+SI]	00	000	00	08	10	18	20	28	30	38
[BX+DI]		001	01	09	11	19	21	29	31	39
[BP+SI]		010	02	0A	12	1A	22	2A	32	3A
[BP+DI]		011	03	0B	13	1B	23	2B	33	3B
[SI]		100	04	0C	14	1C	24	2C	34	3C
[DI]		101	05	0D	15	1D	25	2D	35	3D
disp16		110	06	0E	16	1E	26	2E	36	3E
[BX]		111	07	0F	17	1F	27	2F	37	3F
[BX+SI]+disp8	01	000	40	48	50	58	60	68	70	78
[BX+DI]+disp8		001	41	49	51	59	61	69	71	79
[BP+SI]+disp8		010	42	4A	52	5A	62	6A	72	7A
[BP+DI]+disp8		011	43	4B	53	5B	63	6B	73	7B
[SI]+disp8		100	44	4C	54	5C	64	6C	74	7C
[DI]+disp8		101	45	4D	55	5D	65	6D	75	7D
[BP]+disp8		110	46	4E	56	5E	66	6E	76	7E
[BX]+disp8		111	47	4F	57	5F	67	6F	77	7F
[BX+SI]+disp16	10	000	80	88	90	98	A0	A8	B0	B8
[BX+DI]+disp16		001	81	89	91	99	A1	A9	B1	B9
[BP+SI]+disp16		010	82	8A	92	9A	A2	AA	B2	BA

[BP+DI]+disp16	011	83	8B	93	9B	A3	AB	B3	BB	
[SI]+disp16	100	84	8C	94	9C	A4	AC	B4	BC	
[DI]+disp16	101	85	8D	95	9D	A5	AD	B5	BD	
[BP]+disp16	110	86	8E	96	9E	A6	AE	B6	BE	
[BX]+disp16	111	87	8F	97	9F	A7	AF	B7	BF	
EAX/AX/AL	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH		111	C7	CF	D7	DF	E7	EF	F7	FF

#### Notes:

1. **disp8** denotes an 8-bit displacement following the ModR/M byte, to be sign-extended and added to the index.
2. **disp16** denotes a 16-bit displacement following the ModR/M byte, to be added to the index. Default segment register is SS for the effective addresses containing a BP index, DS for other effective addresses.

-----

## 32-Bit Addressing Forms with the ModR/M Byte

r8(/r)	AL	CL	DL	BL	AH	CH	DH	BH	
r16(/r)	AX	CX	DX	BX	SP	BP	SI	DI	
r32(/r)	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI	
/digit (Opcode)	0	1	2	3	4	5	6	7	
REG =	000	001	010	011	100	101	110	111	

Effective Address	MOD	R/M	MODR/M Values in Hexadecimal							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--] [--]		100	04	0C	14	1C	24	2C	34	3C
disp32		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
disp8[EAX]	01	000	40	48	50	58	60	68	70	78
disp8[ECX]		001	41	49	51	59	61	69	71	79
disp8[EDX]		010	42	4A	52	5A	62	6A	72	7A
disp8[EBX]		011	43	4B	53	5B	63	6B	73	7B
disp8[--] [--]		100	44	4C	54	5C	64	6C	74	7C
disp8[EBP]		101	45	4D	55	5D	65	6D	75	7D
disp8[ESI]		110	46	4E	56	5E	66	6E	76	7E
disp8[EDI]		111	47	4F	57	5F	67	6F	77	7F
disp32[EAX]	10	000	80	88	90	98	A0	A8	B0	B8
disp32[ECX]		001	81	89	91	99	A1	A9	B1	B9
disp32[EDX]		010	82	8A	92	9A	A2	AA	B2	BA
disp32[EBX]		011	83	8B	93	9B	A3	AB	B3	BB
disp32[--] [--]		100	84	8C	94	9C	A4	AC	B4	BC
disp32[EBP]		101	85	8D	95	9D	A5	AD	B5	BD
disp32[ESI]		110	86	8E	96	9E	A6	AE	B6	BE
disp32[EDI]		111	87	8F	97	9F	A7	AF	B7	BF

EAX/AX/AL	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH		111	C7	CF	D7	DF	E7	EF	F7	FF

#### Notes:

1. [--][--] means a SIB follows the ModR/M byte.
2. **disp8** denotes an 8-bit displacement following the SIB byte, to be sign-extended and added to the index.  
**disp32** denotes a 32-bit displacement following the SIB byte, to be added to the index.

-----

## 32-Bit Addressing Forms with the SIB Byte

		EAX	ECX	EDX	EBX	ESP	[*]	ESI	EDI
r32		0	1	2	3	4	5	6	7
Base =		000	001	010	011	100	101	110	111
Base =									

Scaled Index	SS	Index	SIB Values in Hexadecimal							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	99	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7

[EBP*8]	101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]	110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]	111	F8	F9	FA	FB	FC	FD	FE	FF

#### Notes:

[\*] means a disp32 with no base if MOD is 00, [EBP] otherwise. This provides the following addressing modes:

disp32[index] (MOD=00)  
disp8[EBP][index] (MOD=01)  
disp32[EBP][index] (MOD=10)

## How to Read the Instruction Set Pages

The following sections describe how to interpret the description pages for each instruction listed in the [Intel Instruction Set](#) section. Each instruction family is introduced by a descriptive heading such as the following:

#### CMC-Complement Carry Flag

Each instruction family may be accompanied by descriptive sections labeled as follows: [Details Table](#), [Operation](#), [Flags Affected](#), [Protected Mode Exceptions](#), [Real Address Mode Exceptions](#), [Virtual-8086 Mode Exceptions](#), and optionally, a **Notes** section. The following sections explain the notational conventions and abbreviations used in these paragraphs of the instruction descriptions.

## Details Table

For each instruction family, a table is given to list the details of each individual instruction. The following is an example of this table:

Encoding	Instruction	0	1	2	3	4	5	Description
F5	CMC	2	2	2	2	2	2	Complement carry flag

The [Encoding](#), [Instruction](#), and [Description](#) columns are described in the following sections. The columns labeled **0**, **1**, **2**, **3**, **4**, and **5** are collectively described in the [Clocks](#) section. A timing entry appearing in one of these columns is an indication that the associated processor implements that particular instruction variation. The column names are abbreviated, and are described as follows:

0	The 8088/8086/8087 Processor Family
1	The 80186/8087 Processor Family
2	The 80286/80287 Processor Family
3	The 80386/80387 Processor Family
4	The 80486 Processor Family
5	The Pentium Processor Family

## Encoding Column

The "Encoding" column gives the complete object code produced for each form of the instruction. When possible, the codes are given as hexadecimal bytes, in the same order in which they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

- /digit:** (digit is between 0 and 7) indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The **reg** field contains the digit that provides an extension to the instruction's opcode.

- **/r:** indicates that the ModR/M byte of the instruction contains both a register operand and an r/m operand.
- **cb, cw, cd, cp:** a 1-byte (cb), 2-byte (cw), 4-byte (cd) or 6-byte (cp) value following the opcode that is used to specify a code offset and possibly a new value for the code segment register.
- **ib, iw, id:** a 1-byte (ib), 2-byte (iw), or 4-byte (id) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words and doublewords are given with the low-order byte first.
- **+rb, +rw, +rd:** a register code, from 0 through 7, added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte. The codes are-

rb		rw		rd	
AL	= 0	AX	= 0	EAX	= 0
CL	= 1	CX	= 1	ECX	= 1
DL	= 2	DX	= 2	EDX	= 2
BL	= 3	BX	= 3	EBX	= 3
AH	= 4	SP	= 4	ESP	= 4
CH	= 5	BP	= 5	EBP	= 5
DH	= 6	SI	= 6	ESI	= 6
BH	= 7	DI	= 7	EDI	= 7

- **+i:** used in floating-point instructions when one of the operands is ST(i) from the FPU register stack. The number i (which can range from 0 to 7) is added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte.

-----

## Instruction Column

The "Instruction" column gives the syntax of the instruction statement as it would appear in an assembler program. The following is a list of the symbols used to represent operands in the instruction statements:

-----

### rel8

A relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction.

-----

### rel16

A relative address in the range from 32768 bytes before the end of the instruction to 32767 bytes after the end of the instruction. Applies to instructions with an operand-size attribute of 16 bits, and must be within the same code segment as the instruction assembled.

-----

### rel32

A relative address in the range from 2147483648 bytes before the end of the instruction to 2147483647 bytes after the end of the instruction. Applies to instructions with an operand-size attribute of 32 bits, and must be within the same code segment as the instruction assembled.

---

## ptr16:16

A far pointer, typically in a code segment different from that of the instruction. The notation **16:16** indicates that the value of the pointer has two parts. The value to the left of the colon is a 16-bit selector or value destined for the code segment register. The value to the right reflects the operand-size attribute of the instruction (16 bits) and corresponds to the offset within the destination segment.

---

## ptr16:32

A far pointer, typically in a code segment different from that of the instruction. The notation **16:32** indicates that the value of the pointer has two parts. The value to the left of the colon is a 16-bit selector or value destined for the code segment register. The value to the right reflects the operand-size attribute of the instruction (32 bits) and corresponds to the offset within the destination segment.

---

## r8

One of the byte registers AL, CL, DL, BL, AH, CH, DH, or BH.

---

## r16

One of the word registers AX, CX, DX, BX, SP, BP, SI, or DI.

---

## r32

One of the doubleword registers EAX, ECX, EDX, EBX, ESP, EBP, ESI, or EDI.

---

## imm8

An immediate byte value. **imm8** is a signed number between -128 and +127 inclusive. For instructions in which **imm8** is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value.

---

## imm16

An immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between -32768 and +32767 inclusive.

---

## imm32

An immediate doubleword value used for instructions whose operand-size attribute is 32-bits. It allows the use of a number between +2147483647 and -2147483648 inclusive.

-----

## r/m8

A one-byte operand that is either the contents of a byte register (AL, BL, CL, DL, AH, BH, CH, DH), or a byte from memory.

-----

## r/m16

A word register or memory operand used for instructions whose operand-size attribute is 16 bits. The word registers are: AX, BX, CX, DX, SP, BP, SI, DI. The contents of memory are found at the address provided by the effective address computation.

-----

## r/m32

A doubleword register or memory operand used for instructions whose operand-size attribute is 32-bits. The doubleword registers are: EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI. The contents of memory are found at the address provided by the effective address computation.

-----

## r/m64

A quadword register or memory operand used for instructions whose operand-size attribute is 64-bits. The reg/opcode field represents the opcode. The contents of memory are found at the address provided by the effective address computation.

-----

## m

A 16 or 32-bit memory operand.

-----

## m8

A memory byte addressed by DS:[E]SI or ES:[E]DI (used only by string instructions).

-----

## m16

A memory word addressed by DS:[E]SI or ES:[E]DI (used only by string instructions).

-----

## m32

A memory doubleword addressed by DS:[E]SI or ES:[E]DI (used only by string instructions).

-----

## m16:16

A memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset.

-----

## m16:32

A memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset.

-----

## m16&16

A memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. Used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices.

-----

## m16&32

A memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. Used by the LIDT and LGDT to provide a word with which to load the limit field, and a doubleword with which to load the base field of the corresponding Global and Interrupt Descriptor Table Registers.

-----

## m32&32

A memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. Used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices.

-----

## moffs8

The memory offset of a BYTE variable used in some variants of the MOV instruction. The actual address is given by a simple offset relative to the segment base. No ModR/M byte is used in the instruction. The address-size attribute of the instruction determines the size of the offset data.

-----

## moffs16

The memory offset of a WORD variable used in some variants of the MOV instruction. The actual address is given by a simple offset relative

to the segment base. No ModR/M byte is used in the instruction. The address-size attribute of the instruction determines the size of the offset data.

-----

## moffs32

The memory offset of a DWORD variable used in some variants of the MOV instruction. The actual address is given by a simple offset relative to the segment base. No ModR/M byte is used in the instruction. The address-size attribute of the instruction determines the size of the offset data.

-----

## Sreg

A segment register. The segment register bit assignments are ES=0, CS=1, SS=2, DS=3, FS=4, and GS=5.

-----

## m32real

A single-precision floating-point operand in memory.

-----

## m64real

A double-precision floating-point operand in memory.

-----

## m80real

An extended-precision floating-point operand in memory.

-----

## m80bcd

A 80 byte binary-coded decimal operand in memory.

-----

## m16int

A word integer operand in memory. Used in some floating-point instructions.

-----

## m32int

A short integer operand in memory. Used in some floating-point instructions.

-----

## m64int

A long integer operand in memory. Used in some floating-point instructions.

-----

## m14byte

A 14-byte floating-point operand in memory.

-----

## m28byte

A 28-byte floating-point operand in memory.

-----

## m94byte

A 94-byte floating-point operand in memory.

-----

## m108byte

A 108-byte floating-point operand in memory.

-----

## ST or ST(0)

Top element of the FPU register stack.

-----

## ST(i)

*i*th element from the top of the FPU register stack. (i=0..7)

-----

## Clocks Columns

Each "Clocks" column gives the approximate number of clock cycles the instruction takes to execute on that particular processor. The clock

count calculations makes the following assumptions:

- Data and instruction accesses hit in the cache.
- The target of a jump instruction is in the cache.
- No invalidate cycles contend with the instruction for use of the cache.
- Page translation hits in the TLB.
- Memory operands are aligned.
- Effective address calculations use a base register which is not the destination register of the preceding instruction.
- No exceptions are detected during execution.
- There are no write-buffer delays.

The following symbols are used in the clock count specifications:

- **n**, which represents a number of repetitions.
- **m**, which represents the number of components in the next instruction executed, where the entire displacement (if any) counts as one component, the entire immediate data (if any) counts as one component, and every other byte of the instruction and prefix(es) each counts as one component.
- **pm:**, a clock count that applies when the instruction executes in Protected Mode. **pm:** is not given when the clock counts are the same for Protected and Real Address Modes.

When an exception occurs during the execution of an instruction and the exception handler is in another task, the instruction execution time is increased by the number of clocks to effect a task switch. This parameter depends on several factors:

- The type of TSS used to represent the new task (32 bit TSS or 16 bit TSS).
- Whether the current task is in V86 mode.
- Whether the new task is in V86 mode.
- Whether accesses hit in the cache.
- Whether a task gate on an interrupt/trap gate is used.

The following table summarizes the task switch times for exceptions, assuming cache hits and the use of task gates.

#### Task Switch Times for Exceptions

OLD TASK	NEW TASK		
	TO 32 BIT TSS	TO 16 BIT TSS	TO VM TSS
VM/32 bit/16 bit TSS	85	87	71

## Description Column

The "Description" column following the "Clocks" columns briefly explains the various forms of the instruction. The "Operation" and "Description" sections contain more details of the instruction's operation.

## Description

The "Description" section contains further explanation of the instruction's operation.

# Operation

The "Operation" section contains an algorithmic description of the instruction which uses a notation similar to the Algol or Pascal language. The algorithms are composed of the following elements:

- Comments are enclosed within the symbol pairs "(" and ")".
- Compound statements are enclosed between the keywords of the "if" statement (IF, THEN, ELSE, FI) or of the "do" statement (DO, OD), or of the "case" statement (CASE ... OF, ESAC).
- Execution continues until the END statement is encountered.
- A register name implies the contents of the register. A register name enclosed in brackets implies the contents of the location whose address is contained in that register. For example, ES:[DI] indicates the contents of the location whose ES segment relative address is in register DI. [SI] indicates the contents of the address contained in register SI relative to SI's default segment (DS) or overridden segment.
- Brackets are also used for memory operands, where they mean that the contents of the memory location is a segment-relative offset. For example, [SRC] indicates that the contents of the source operand is a segment-relative offset.
- A ` B; indicates that the value of B is assigned to A.
- The symbols =, <>, >, and % are relational operators used to compare two values, meaning equal, not equal, greater or equal, less or equal, respectively. A relational expression such as A = B is TRUE if the value of A is equal to B; otherwise it is FALSE.
- A \* B indicates that the value of A is multiplied by the value of B.

The following identifiers are used in the algorithmic descriptions:

- **OperandSize** represents the operand-size attribute of the instruction, which is either 16 or 32 bits. **AddressSize** represents the address-size attribute, which is either 16 or 32 bits. For example,

```
IF instruction = CMPSW
THEN OperandSize ` 16;
ELSE
  IF instruction = CMPSD
  THEN OperandSize ` 32;
  FI;
FI;
```

indicates that the operand-size attribute depends on the form of the CMPS instruction used. Refer to the explanation of address-size and operand-size attributes at the beginning of this chapter for general guidelines on how these attributes are determined.

- **StackAddrSize** represents the stack address-size attribute associated with the instruction, which has a value of 16 or 32 bits, as explained earlier in the chapter.
- **SRC** represents the source operand. When there are two operands, SRC is the one on the right.
- **DEST** represents the destination operand. When there are two operands, DEST is the one on the left.
- **LeftSRC, RightSRC** distinguishes between two operands when both are source operands.
- **eSP** represents either the SP register or the ESP register depending on the setting of the B-bit for the current stack segment.

The following functions are used in the algorithmic descriptions:

- **Truncate to 16 bits(value)** reduces the size of the value to fit in 16 bits by discarding the uppermost bits as needed.
- **Addr(operand)** returns the effective address of the operand (the result of the effective address calculation prior to adding the segment base).
- **ZeroExtend(value)** returns a value zero-extended to the operand-size attribute of the instruction. For example, if OperandSize = 32, ZeroExtend of a byte value of -10 converts the byte from F6H to doubleword with hexadecimal value 000000F6H. If the value passed to ZeroExtend and the operand-size attribute are the same size, ZeroExtend returns the value unaltered.
- **SignExtend(value)** returns a value sign-extended to the operand-size attribute of the instruction. For example, if OperandSize = 32, SignExtend of a byte containing the value -10 converts the byte from F6H to a doubleword with hexadecimal value FFFFFFF6H. If the value passed to SignExtend and the operand-size attribute are the same size, SignExtend returns the value unaltered.

- Push(value)** pushes a value onto the stack. The number of bytes pushed is determined by the operand-size attribute of the instruction. The action of Push is as follows:  
 IF StackAddrSize = 16  
 THEN  
   IF OperandSize = 16  
   THEN  
     SP ` SP -2;  
     SS:[SP] ` value; (\* 2 bytes assigned starting at byte address in SP \*)  
   ELSE (\* OperandSize = 32 \*)  
     SP ` SP -4;  
     SS:[SP] ` value; (\* 4 bytes assigned starting at byte address in SP \*)  
   FI;  
 ELSE (\* StackAddrSize = 32 \*)  
   IF OperandSize = 16  
   THEN  
     ESP ` ESP -2;  
     SS:[ESP] ` value; (\* 2 bytes assigned starting at byte address in ESP \*)  
   ELSE (\* OperandSize = 32 \*)  
     ESP ` ESP -4;  
     SS:[ESP] ` value; (\* 4 bytes assigned starting at byte address in ESP \*)  
   FI;  
 FI;
- Pop(value)** removes the value from the top of the stack and returns it. The statement EAX ` Pop( ); assigns to EAX the 32-bit value that Pop took from the top of the stack. Pop will return either a word or a doubleword depending on the operand-size attribute. The action of Pop is as follows:  
 IF StackAddrSize = 16  
 THEN  
   IF OperandSize = 16  
   THEN  
     ret val ` SS:[SP]; (\* 2-byte value \*)  
     SP ` SP + 2;  
   ELSE (\* OperandSize = 32 \*)  
     ret val ` SS:[SP]; (\* 4-byte value \*)  
     SP ` SP + 4;  
   FI;  
 ELSE (\* StackAddrSize = 32 \*)  
   IF OperandSize = 16  
   THEN  
     ret val ` SS:[ESP]; (\* 2 byte value \*)  
     ESP ` ESP + 2;  
   ELSE (\* OperandSize = 32 \*)  
     ret val ` SS:[ESP]; (\* 4 byte value \*)  
     ESP ` ESP + 4;  
   FI;  
 FI;  
 RETURN(ret val); (\*returns a word or doubleword\*)

Pop ST is used on floating-point instruction pages to mean *pop the FPU register stack*.

- Bit[BitBase, BitOffset]** returns the value of a bit within a bit string, which is a sequence of bits in memory or a register. Bits are numbered from low-order to high-order within registers and within memory bytes. In memory, the two bytes of a word are stored with the low-order byte at the lower address.

If the base operand is a register, the offset can be in the range 0..31. This offset addresses a bit within the indicated register. An example, 'BIT[EAX, 21]' is illustrated in the following figure.

Bit Offset for BIT[EAX,21]

31                      21                                      0

.

.

BITOFFSET=21

If BitBase is a memory address, BitOffset can range from -2 gigabits to 2 gigabits. The addressed bit is numbered (Offset MOD 8) within the byte at address (BitBase + (BitOffset DIV 8)), where DIV is signed division with rounding towards negative infinity, and MOD returns a positive number. This is illustrated in the following figure.

Memory Bit Indexing

7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0

BITBASE+1                      BITBASE                      BITBASE-1  
 .  
 OFFSET=+13

7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0

BITBASE                      BITBASE-1                      BITBASE-2  
 .  
 OFFSET=-11

- I-O-Permission(I-O-Address, width)** returns TRUE or FALSE depending on the I/O permission bitmap and other factors. This function is defined as follows:  
 IF TSS type is 16-bit THEN RETURN FALSE; FI;  
 Ptr` [TSS+66]; (\* fetch bitmap pointer \*)  
 BitStringAddr` SHR (I-O-Address, 3) + Ptr;  
 MaskShift` I-O-Address AND 7;  
 CASE width OF:  
   BYTE: nBitMask` 1;  
   WORD: nBitMask` 3;  
   DWORD: nBitMask` 15;  
 ESAC;  
 mask` SHL (nBitMask, MaskShift);  
 CheckString` [BitStringAddr] AND mask;  
 IF CheckString = 0  
   THEN RETURN (TRUE);  
   ELSE RETURN (FALSE);  
 FI;
- Switch-Tasks** is described in detail in the Intel documentation.

-----

## Flags Affected

Pages describing basic instructions have a "Flags Affected" section the contains a flags information table similar to the following:

OF	DF	IF	SF	ZF	AF	PF	CF
0		*	*	?	*		0

The first row of the table lists the mnemonic identifiers for the various flags. The entries in the second row are filled in according to how the flag is affected by the instruction:

VALUE	MEANING
<blank>	Instruction does not affect flag
0	Instruction clears the flag
1	Instruction sets the flag
?	Instruction's effect on the flag is undefined
*	Instruction modifies the flag (either sets or clears depending on operands)

The following table lists the mnemonic identifier, full name, and purpose of the flags that are applicable to all processor families and that are

are most commonly used from within application-level programs. Not all flags are included in this table; see the Intel documentation for a more complete description of flag usage from within systems-level programs.

MNEMONIC	FLAG NAME	PURPOSE
OF	Overflow	Result exceeds positive or negative limit of number range
DF	Direction	Setting the DF flag causes string instructions to auto-decrement, that is, to process strings from high addresses. Clearing the DF flag causes string instructions to auto-increment, or to process strings from low addresses to high addresses.
IF	Interrupt Enable	Controls the acceptance of external interrupts signalled via the INTR pin.
SF	Sign	Result is negative (less than zero)
ZF	Zero	Result is zero
AF	Auxiliary carry	Carry out of bit position 3 (used for BCD)
PF	Parity	Low byte of result has even parity (even number of set bits)
CF	Carry	Carry out of most significant bit of result

The flags information table is usually followed by a paragraph description of how the flags are affected:

- If a flag is always cleared or always set by the instruction, the value is given (0 or 1) after the flag name. Arithmetic and logical instructions usually assign values to the status flags in a uniform manner. Nonconventional assignments are described in the [Operation](#) section.
- The values of flags listed as "undefined" may be changed by the instruction in an indeterminate manner.

All flags not listed are unchanged by the instruction.

-----

## FPU Flags Affected

The floating-point instruction pages have a section called "FPU Flags Affected," which tells how each instruction can affect the four condition code bits of the FPU status word. These pages contain a condition code information table similar to the following:

C0	C1	C2	C3
?	*	?	?

The first row of the table lists the names of the floating-point condition code flags. The entries in the second row are filled in according to how the flag is affected by the instruction:

VALUE	MEANING
<blank>	Instruction does not affect flag
0	Instruction clears the flag
1	Instruction sets the flag
?	Instruction's effect on the flag is undefined
*	Instruction modifies the flag (either sets or clears depending on operands)

The four FPU condition code bits (C0, C1, C2, and C3) are similar to the flags in a CPU; the processor updates these bits to reflect the outcome of arithmetic operations. The effect of these instructions on the condition code bits is summarized in the following table:

#### Condition Code Interpretation

INSTRUCTION	C0	C3	C2	C1
FCOM, FCOMP, FCOMPP, FTST, FUCOMPP, FICOM, FICOMP	Result of Comparison		Operands is not Comparable	Zero or O/U#
FXAM	Operand class			Sign or O/U#
FPREM, FPREM1	Q2	Q1	0=reduction complete  1=reduction incomplete	Q0 or O/U#
FIST, FBSTP, FRINDINT, FST, FSTP, FADD, FMUL, FDIV, FDIVR, FSUB, FSUBR, FSCALE, FSQRT, FPATAN, F2XM1, FYL2X, FYL2XP1	UNDEFINED			Roundup or O/U#
FPTAN, FSIN, FCOS, FSINCOS	UNDEFINED		0=reduction complete  1=reduction incomplete	Roundup or O/U# (UNDEFINED if C2=1)
FCHS, FABS, FXCH, FINCSTP, FDECSTP, Constant Loads, FXTRACT, FLD, FILD, FBLD, FSTP (ext. real)	UNDEFINED			Zero or O/U#
FLDENV, FRSTOR	Each bit loaded from memory			
FLDCW, FSTENV, FSTCW, FSTSW, FCLEX	UNDEFINED			
FINIT, FSAVE	Zero	Zero	Zero	Zero

#### NOTES:

<b>O/U#</b>	When both IE and SF bits of status word are set, this bit distinguishes between stack overflow (C1=1) and underflow (C1=0).
<b>Reduction</b>	If FPREM and FPREM1 produces a remainder that is less than the modulus, reduction is complete. When reduction is incomplete the value at the top of the stack is a partial remainder, which can be used as input to further reduction. For FPTAN, FSIN, FCOS and FSINCOS, the reduction bit is set if the operand at the top of the stack is too large. In this case, the original operand remains at the top of the stack.
<b>Roundup</b>	When the PE bit of the status word is set, this bit indicates whether the last rounding in the instruction was upward.
<b>UNDEFINED</b>	Do not rely on any specific value in these bits.

The condition code bits are used primarily for conditional branching. The FSTSW AX instruction stores the FPU status word directly into the AX register, allowing these condition codes to be inspected efficiently. The SAHF instruction can copy C3 - C0 directly to the CPU's flag bits to simplify conditional branching. The following table shows the mapping of these bits to the CPU flag bits.

FPU FLAG	IU FLAG
C0	CF

C1	(None)
C2	PF
C3	ZF

# Numeric Exceptions

For floating-point instruction pages, this section lists the exception flags of the FPU status word that each instruction can set. Exceptions are listed in abbreviated form, and are defined as follows:

IS	Invalid operand due to stack overflow/underflow
I	Invalid operand due to other cause
D	Denormalized operand
Z	Divide by zero
O	Numeric overflow
U	Numeric underflow
P	Inexact result (precision)

# Protected Mode Exceptions

This section lists the exceptions that can occur when the instruction is executed in protected mode. The exception names are a pound sign (#) followed by two letters and an optional error code in parentheses. For example, #GP(0) denotes a general protection exception with an error code of 0. The following table associates each two-letter name with the corresponding interrupt number.

## Exceptions

MNEMONIC	INTERRUPT	DESCRIPTION
#UD	6	Invalid opcode
#NM	7	Device not available
#DF	8	Double fault
#TS	10	Invalid TSS
#NP	11	Segment or gate not present
#SS	12	Stack fault
#GP	13	General protection fault
#PF	14	Page fault
#MF	16	Floating-point error
#AC	17	Alignment check

Refer to the Intel documentation for a description of the exceptions and the processor state upon entry to the exception.

Application programmers should consult the documentation provided with their operating systems to determine the actions taken when exceptions occur.

# Real Address Mode Exceptions

Because less error checking is performed by the processor in Real Address Mode, this mode has fewer exception conditions. Refer to the

Intel documentation for further information on these exceptions.

-----

# Virtual-8086 Mode Exceptions

Virtual 8086 tasks provide the ability to simulate Virtual 8086 machines. Virtual 8086 Mode exceptions are similar to those for the 8086 processor, but there are some differences. Refer to the Intel documentation for complete information on Virtual Mode exceptions.

-----

## Instruction Timing

ý

-----

## Instruction Timing

(102-137)+EA

-----

## Instruction Timing

(103-139)+EA

-----

## Instruction Timing

(108-143)+EA

-----

## Instruction Timing

(109-144)+EA

-----

## Instruction Timing

(110-125)+EA

-----

## Instruction Timing

(12-18)+fw

## Instruction Timing

(124-138)+EA

## Instruction Timing

(130-144)+EA

## Instruction Timing

(154-168)+EA

## Instruction Timing

(197-207)+EA

## Instruction Timing

(2-8)+fw

## Instruction Timing

(215-225)+EA

## Instruction Timing

(216-226)+EA

## Instruction Timing

(220-230)+EA

# Instruction Timing

(221-231)+EA

-----

# Instruction Timing

(224-238)+EA

-----

# Instruction Timing

(225-239)+EA

-----

# Instruction Timing

(230-243)+EA

-----

# Instruction Timing

(231-245)+EA

-----

# Instruction Timing

(290-310)+EA

-----

# Instruction Timing

(35-45)+EA

-----

# Instruction Timing

(38-56)+EA

-----

# Instruction Timing

(40-60)+EA

## Instruction Timing

(46-54)+EA

## Instruction Timing

(52-58)+EA

## Instruction Timing

(52-60)+EA

## Instruction Timing

(520-540)+EA

## Instruction Timing

(53-65)+EA

## Instruction Timing

(60-68)+EA

## Instruction Timing

(60-70)+EA

## Instruction Timing

(63-73)+EA

## Instruction Timing

(65-75)+EA

## Instruction Timing

(67-77)+EA

## Instruction Timing

(7-14)+EA

## Instruction Timing

(72-86)+EA

## Instruction Timing

(74-88)+EA

## Instruction Timing

(78-91)+EA

## Instruction Timing

(80-90)+EA

## Instruction Timing

(80-93)+EA

# Instruction Timing

(82-92)+EA

-----

# Instruction Timing

(84-90)+EA

-----

# Instruction Timing

(84-94)+EA

-----

# Instruction Timing

(86-92)+EA

-----

# Instruction Timing

(90-120)+EA

-----

# Instruction Timing

(94-105)+EA

-----

# Instruction Timing

(95-125)+EA

-----

# Instruction Timing

(96-104)+EA

-----

# Instruction Timing

(98-106)+EA

Instruction Timing

0

Instruction Timing

1

Instruction Timing

1-3

Instruction Timing

1/2

Instruction Timing

1/3

Instruction Timing

10

Instruction Timing

10+3n

Instruction Timing

10+m

## Instruction Timing

10, pm:4\*/24\*\*

## Instruction Timing

10-15

## Instruction Timing

10-16

## Instruction Timing

10-17

## Instruction Timing

10-18

## Instruction Timing

10/13

## Instruction Timing

101-112/ (107-118)+EA

## Instruction Timing

102-137

# Instruction Timing

103-139

-----

# Instruction Timing

108-143

-----

# Instruction Timing

109-144

-----

# Instruction Timing

11

-----

# Instruction Timing

11+fw

-----

# Instruction Timing

11+m

-----

# Instruction Timing

11, pm:5\*/25\*\*

-----

# Instruction Timing

11,pm:23

-----

# Instruction Timing

11-17

## Instruction Timing

11/18+EA

## Instruction Timing

110-125

## Instruction Timing

118-133/ (124-139)+EA

## Instruction Timing

12

## Instruction Timing

$12+4(n-1)$

## Instruction Timing

$12+m$ , pm:27+m

## Instruction Timing

12, pm:6\*/26\*\*

## Instruction Timing

12, pm:9\*/26\*\*, vm:24

## Instruction Timing

12-18

## Instruction Timing

12-23

## Instruction Timing

12-42/13-42

## Instruction Timing

120-127

## Instruction Timing

120-538

## Instruction Timing

121-128

## Instruction Timing

122-129

## Instruction Timing

122-771

# Instruction Timing

123-772

-----

# Instruction Timing

124-138

-----

# Instruction Timing

128-154/ (134-160)+EA

-----

# Instruction Timing

13

-----

# Instruction Timing

13, pm:10\*/27\*\*, vm:25

-----

# Instruction Timing

13, pm:7\*/27\*\*

-----

# Instruction Timing

13,pm:18

-----

# Instruction Timing

13,pm:26

-----

# Instruction Timing

13,pm:33

## Instruction Timing

13-16

## Instruction Timing

13-18

## Instruction Timing

13-26

## Instruction Timing

13-42

## Instruction Timing

13-57

## Instruction Timing

13/16

## Instruction Timing

13/18

## Instruction Timing

13/26

## Instruction Timing

13/42

## Instruction Timing

130-144

## Instruction Timing

130-145

## Instruction Timing

131,pm:120

## Instruction Timing

## Instruction Timing

134-148

## Instruction Timing

135-141

## Instruction Timing

136-140

# Instruction Timing

14

-----

# Instruction Timing

14, pm:8\*/28\*\*

-----

# Instruction Timing

14, pm:8\*/28\*\*, vm:27

-----

# Instruction Timing

14,pm:17

-----

# Instruction Timing

14,pm:25

-----

# Instruction Timing

14,pm:33

-----

# Instruction Timing

14/16

-----

# Instruction Timing

14/17

-----

# Instruction Timing

140-279

## Instruction Timing

144-162/ (150-168)+EA

## Instruction Timing

148-154

## Instruction Timing

15

## Instruction Timing

15+2n

## Instruction Timing

15+4(n-1)

## Instruction Timing

15+fw

## Instruction Timing

15+m, pm:26+m

## Instruction Timing

15, pm:9\*/29\*\*

## Instruction Timing

15,pm:25

## Instruction Timing

15-17

## Instruction Timing

15-190

## Instruction Timing

15-21

## Instruction Timing

15-22

## Instruction Timing

15/16

## Instruction Timing

154-168

## Instruction Timing

16

# Instruction Timing

16+EA

-----

# Instruction Timing

16+fw

-----

# Instruction Timing

16, pm:11\*/31\*\*, vm:29

-----

# Instruction Timing

16-126

-----

# Instruction Timing

16-20

-----

# Instruction Timing

16-22

-----

# Instruction Timing

16-50

-----

# Instruction Timing

16-64

-----

# Instruction Timing

16/16

## Instruction Timing

16/21+EA

## Instruction Timing

16/29

## Instruction Timing

16/4

## Instruction Timing

165-184/ (171-190)+EA

## Instruction Timing

169

## Instruction Timing

17

## Instruction Timing

17+3n

## Instruction Timing

17+EA

## Instruction Timing

17+fw

## Instruction Timing

17+m; pm:34+m

## Instruction Timing

17, pm:10\*/32\*\*, vm:30

## Instruction Timing

17, pm:10\*32\*\*, vm:30

## Instruction Timing

17,pm:19

## Instruction Timing

17,pm:31

## Instruction Timing

17-137

## Instruction Timing

17-173

# Instruction Timing

17-22

-----

# Instruction Timing

17-23

-----

# Instruction Timing

17-24

-----

# Instruction Timing

17/19

-----

# Instruction Timing

17/20

-----

# Instruction Timing

17/5

-----

# Instruction Timing

171-326

-----

# Instruction Timing

172-176

-----

# Instruction Timing

## Instruction Timing

177/182

## Instruction Timing

178

## Instruction Timing

17;pm:20

## Instruction Timing

18

## Instruction Timing

18+m, pm:32+m

## Instruction Timing

18-124

## Instruction Timing

18-24

## Instruction Timing

18/6

## Instruction Timing

180

## Instruction Timing

180-186

## Instruction Timing

180/185

## Instruction Timing

183

## Instruction Timing

18;pm:20

## Instruction Timing

19

## Instruction Timing

19+TS

## Instruction Timing

19-32

# Instruction Timing

19/20

-----

# Instruction Timing

19/5

-----

# Instruction Timing

191-497

-----

# Instruction Timing

193-203

-----

# Instruction Timing

194-204

-----

# Instruction Timing

194-809

-----

# Instruction Timing

196-329

-----

# Instruction Timing

197-207

-----

# Instruction Timing

## Instruction Timing

2

## Instruction Timing

2+EA

## Instruction Timing

2+fw

## Instruction Timing

2-8

## Instruction Timing

2/15+EA

## Instruction Timing

2/2

## Instruction Timing

2/3

## Instruction Timing

2/3,pm:2

## Instruction Timing

2/4

## Instruction Timing

2/5

## Instruction Timing

2/5,pm:17/19

## Instruction Timing

2/5,pm:18/19

## Instruction Timing

2/6

## Instruction Timing

2/7

## Instruction Timing

2/8+EA

## Instruction Timing

2/9+EA

# Instruction Timing

20

-----

# Instruction Timing

20+TS

-----

# Instruction Timing

20-24

-----

# Instruction Timing

20-31

-----

# Instruction Timing

20-35

-----

# Instruction Timing

20-55

-----

# Instruction Timing

20-70

-----

# Instruction Timing

20/21

-----

# Instruction Timing

200-273

## Instruction Timing

2000+

## Instruction Timing

205-215

## Instruction Timing

21

## Instruction Timing

21-30

## Instruction Timing

21-303

## Instruction Timing

21/24

## Instruction Timing

21/46

## Instruction Timing

211-476

## Instruction Timing

215-225

## Instruction Timing

216-226

## Instruction Timing

22

## Instruction Timing

22+m; pm:38+m

## Instruction Timing

22,pm:38

## Instruction Timing

22-103

## Instruction Timing

22-111

## Instruction Timing

22-24

# Instruction Timing

22/25

-----

# Instruction Timing

220-230

-----

# Instruction Timing

221-231

-----

# Instruction Timing

224-238

-----

# Instruction Timing

225-239

-----

# Instruction Timing

23

-----

# Instruction Timing

23-27

-----

# Instruction Timing

23-31

-----

# Instruction Timing

## Instruction Timing

230-243

## Instruction Timing

231-245

## Instruction Timing

24

## Instruction Timing

24+EA

## Instruction Timing

24-25

## Instruction Timing

24-32

## Instruction Timing

24/24

## Instruction Timing

25

## Instruction Timing

25-33

## Instruction Timing

25/26

## Instruction Timing

25/28

## Instruction Timing

250-800

## Instruction Timing

257-354

## Instruction Timing

257-547

## Instruction Timing

26

## Instruction Timing

26-34

# Instruction Timing

266-275

-----

# Instruction Timing

27

-----

# Instruction Timing

27-35

-----

# Instruction Timing

27-55

-----

# Instruction Timing

27/28

-----

# Instruction Timing

28

-----

# Instruction Timing

28-34

-----

# Instruction Timing

29-34

-----

# Instruction Timing

29-37

## Instruction Timing

29-57

## Instruction Timing

290-310

## Instruction Timing

292-365

## Instruction Timing

3

## Instruction Timing

3+fw

## Instruction Timing

3/1

## Instruction Timing

3/12

## Instruction Timing

3/15+EA

## Instruction Timing

3/16+EA

## Instruction Timing

3/3

## Instruction Timing

3/4

## Instruction Timing

3/5

## Instruction Timing

3/6

## Instruction Timing

3/7

## Instruction Timing

3/8

## Instruction Timing

3/9

# Instruction Timing

3/9+EA

-----

# Instruction Timing

30

-----

# Instruction Timing

30-32

-----

# Instruction Timing

30-38

-----

# Instruction Timing

30-45

-----

# Instruction Timing

30-540

-----

# Instruction Timing

308

-----

# Instruction Timing

31

-----

# Instruction Timing

310-630

## Instruction Timing

314-487

## Instruction Timing

32

## Instruction Timing

32-38

## Instruction Timing

32-57

## Instruction Timing

33

## Instruction Timing

33+fw

## Instruction Timing

35-45

## Instruction Timing

36

## Instruction Timing

37+EA

## Instruction Timing

37, pm16:32, pm32:33

## Instruction Timing

38

## Instruction Timing

38-36

## Instruction Timing

38-48

## Instruction Timing

38-56

## Instruction Timing

38/41

## Instruction Timing

39

## Instruction Timing

4

-----

## Instruction Timing

$4+5n$

-----

## Instruction Timing

$4/1$

-----

## Instruction Timing

$4/10+EA$

-----

## Instruction Timing

$4/17+EA$

-----

## Instruction Timing

$4/3$

-----

## Instruction Timing

$4/5$

-----

## Instruction Timing

$4/9$

-----

## Instruction Timing

4/pm:8

## Instruction Timing

40

## Instruction Timing

40-50

## Instruction Timing

40-60

## Instruction Timing

40/40

## Instruction Timing

41

## Instruction Timing

41+TS

## Instruction Timing

42

## Instruction Timing

42+TS

## Instruction Timing

42-52

## Instruction Timing

43

## Instruction Timing

43+TS

## Instruction Timing

43+m, pm:31+m

## Instruction Timing

43/44

## Instruction Timing

44

## Instruction Timing

44, pm:34

## Instruction Timing

## Instruction Timing

45

-----

## Instruction Timing

45+m

-----

## Instruction Timing

45-52

-----

## Instruction Timing

45-55

-----

## Instruction Timing

46

-----

## Instruction Timing

46-54

-----

## Instruction Timing

48-58

-----

## Instruction Timing

49+m

-----

## Instruction Timing

## Instruction Timing

5+TS

## Instruction Timing

5+ts

## Instruction Timing

5,pm:20

## Instruction Timing

5/11+EA

## Instruction Timing

5/3

## Instruction Timing

5/5

## Instruction Timing

5/6

## Instruction Timing

5/8

## Instruction Timing

512-534

## Instruction Timing

52-58

## Instruction Timing

52-60

## Instruction Timing

520-540

## Instruction Timing

53

## Instruction Timing

53-65

## Instruction Timing

55

## Instruction Timing

56-63

# Instruction Timing

56-67

-----

# Instruction Timing

57-72

-----

# Instruction Timing

57-82

-----

# Instruction Timing

58-83

-----

# Instruction Timing

6

-----

# Instruction Timing

6,5

-----

# Instruction Timing

6,pm:4

-----

# Instruction Timing

6-103/7-104

-----

# Instruction Timing

6-12

## Instruction Timing

6-34/6-35

## Instruction Timing

6-42/6-43

## Instruction Timing

6-42/7-43

## Instruction Timing

6/12

## Instruction Timing

6/13

## Instruction Timing

6/7(=); 6/10(!=)

## Instruction Timing

6/8

## Instruction Timing

60

## Instruction Timing

60-68

## Instruction Timing

60-70

## Instruction Timing

61-65

## Instruction Timing

61-82

## Instruction Timing

63-73

## Instruction Timing

65-75

## Instruction Timing

66-80

## Instruction Timing

67-77

# Instruction Timing

67-86

-----

# Instruction Timing

68

-----

# Instruction Timing

7

-----

# Instruction Timing

7+fw

-----

# Instruction Timing

7+m

-----

# Instruction Timing

7+m,10+m

-----

# Instruction Timing

7+m/10+m

-----

# Instruction Timing

7+m/3

-----

# Instruction Timing

## Instruction Timing

7,pm:21

## Instruction Timing

7,pm:22

## Instruction Timing

7,pm:25

## Instruction Timing

7-14

## Instruction Timing

7-24/9-26

## Instruction Timing

7-39/7-40

## Instruction Timing

7-71/7-72

## Instruction Timing

7/11

## Instruction Timing

7/13

## Instruction Timing

7/3

## Instruction Timing

7/4

## Instruction Timing

7/5

## Instruction Timing

7/8

## Instruction Timing

70

## Instruction Timing

70-100

## Instruction Timing

70-103

## Instruction Timing

70-138

-----

## Instruction Timing

70-76

-----

## Instruction Timing

70-77/(76-83)+EA

-----

## Instruction Timing

700-1000

-----

## Instruction Timing

71

-----

## Instruction Timing

71-75

-----

## Instruction Timing

71-83

-----

## Instruction Timing

72-167

-----

## Instruction Timing

72-84

## Instruction Timing

72-86

## Instruction Timing

73

## Instruction Timing

74-155

## Instruction Timing

74-88

## Instruction Timing

75-105

## Instruction Timing

75-85

## Instruction Timing

76-87

## Instruction Timing

78-91

## Instruction Timing

79-93

## Instruction Timing

8

## Instruction Timing

$8+4 \text{ per bit}/(20+4 \text{ per bit})+EA$

## Instruction Timing

8,4

## Instruction Timing

8-20

## Instruction Timing

8-25/10-27

## Instruction Timing

8-30/9-31

## Instruction Timing

8/4

# Instruction Timing

8/int+32

-----

# Instruction Timing

80-90

-----

# Instruction Timing

80-90/(86-96)+EA

-----

# Instruction Timing

80-93

-----

# Instruction Timing

80-97

-----

# Instruction Timing

80-98/(86-104)+EA

-----

# Instruction Timing

82

-----

# Instruction Timing

82-92

-----

# Instruction Timing

82-95

## Instruction Timing

83

## Instruction Timing

83-87

## Instruction Timing

84-86

## Instruction Timing

84-90

## Instruction Timing

84-94

## Instruction Timing

85-89

## Instruction Timing

86+4x

## Instruction Timing

86-92

## Instruction Timing

88

## Instruction Timing

89

## Instruction Timing

9

## Instruction Timing

9+fw

## Instruction Timing

9+m/5

## Instruction Timing

9, pm:6\*/24\*\*, vm:22

## Instruction Timing

9,pm:6

## Instruction Timing

9-12

# Instruction Timing

9-14

-----

# Instruction Timing

9-14/12-17

-----

# Instruction Timing

9-16

-----

# Instruction Timing

9-20

-----

# Instruction Timing

9-22

-----

# Instruction Timing

9-22/12-25

-----

# Instruction Timing

9-38

-----

# Instruction Timing

9-38/12-41

-----

# Instruction Timing

9/10

## Instruction Timing

9/6

## Instruction Timing

9/9

## Instruction Timing

$90+4x+m$

## Instruction Timing

90-120

## Instruction Timing

900-1100

## Instruction Timing

91

## Instruction Timing

94

## Instruction Timing

94-105

## Instruction Timing

95-125

## Instruction Timing

95-185

## Instruction Timing

96-104

## Instruction Timing

98-106

## Instruction Timing

TS

## Instruction Timing

TS+10

## Instruction Timing

TS+32

## Instruction Timing

hit=12, nohit=11

# Instruction Timing

pm:10/11

-----

# Instruction Timing

pm:20/21

-----

# Instruction Timing

pm:21+ts

-----

# Instruction Timing

pm:22

-----

# Instruction Timing

pm:35

-----

# Instruction Timing

pm:37+ts

-----

# Instruction Timing

pm:37+tx

-----

# Instruction Timing

pm:44

-----

# Instruction Timing

pm:45+2x

## Instruction Timing

pm:52+m

## Instruction Timing

pm:56+m

## Instruction Timing

pm:69

## Instruction Timing

pm:7

## Instruction Timing

pm:77+4x

## Instruction Timing

pm:86+m

## Instruction Timing

pm:90+m

## Instruction Timing

pm:94+ 4x+m

## Instruction Timing

pm:98+ 4x+m

## Instruction Timing

ts

## Instruction Timing

xm16:75, xm32:95; pm:70

## Instruction Timing

(197-207)+EA+fw

## Instruction Timing

(40-50)+EA+fw

## Instruction Timing

22+16(n-1)

## Instruction Timing

22-25

## Instruction Timing

22-25/29-32

## Instruction Timing

$3+5n$

-----

## Instruction Timing

$33-35$

-----

## Instruction Timing

$5+n/17+n$

-----

## Instruction Timing

$8+9n$

-----

## Instruction Timing

$(10-16)+fw$

-----

## Instruction Timing

$(103-104)+fw$

-----

## Instruction Timing

$(154,pm=143)+fw$

-----

## Instruction Timing

$(205-215)+fw$

-----

## Instruction Timing

(375-376)+fw

## Instruction Timing

(40-50)+EA

## Instruction Timing

(40-50)+fw

## Instruction Timing

(67,pm=56)+fw

## Instruction Timing

(xm16/32=127/151,pm=124)+fw

## Instruction Timing

(xm16/32=50/48,pm16/32=49/50)+fw

## Instruction Timing

10/11

## Instruction Timing

103-104

## Instruction Timing

13+fw

## Instruction Timing

154,pm=143

## Instruction Timing

375-376

## Instruction Timing

4,pm=3

## Instruction Timing

67,pm=56

## Instruction Timing

??

## Instruction Timing

xm16/32=127/151,pm=124

## Instruction Timing

xm16/32=50/48,pm16/32=49/50

## Instruction Timing

119

# Instruction Timing

167

-----

# Instruction Timing

1:119,0:3

-----

# Instruction Timing

1:13,0:4

-----

# Instruction Timing

1:168,0:3

-----

# Instruction Timing

1:19+TS,0:4

-----

# Instruction Timing

1:24,0:3

-----

# Instruction Timing

1:27,0:4

-----

# Instruction Timing

1:28,0:3

-----

# Instruction Timing

1:35,0:3

## Instruction Timing

1:39+TS,0:3

## Instruction Timing

1:41,0:3

## Instruction Timing

1:44,0:4

## Instruction Timing

1:46,0:3

## Instruction Timing

1:53,0:4

## Instruction Timing

1:56,0:4

## Instruction Timing

1:59,0:3

## Instruction Timing

1:73,0:3

## Instruction Timing

1:79,0:3

## Instruction Timing

1:84,0:3

## Instruction Timing

1:99,0:3

## Instruction Timing

1:TS,0:3

## Instruction Timing

23+TS

## Instruction Timing

37

## Instruction Timing

37+TS

## Instruction Timing

48

## Instruction Timing

51

-----

## Instruction Timing

52

-----

## Instruction Timing

56

-----

## Instruction Timing

59

-----

## Instruction Timing

78

-----

## Instruction Timing

86

-----

## Instruction Timing

99

-----

## Instruction Timing

11+3(E)CX, pm:8+3(E)CX\*1/25+3(E)CX\*2

-----

## Instruction Timing

13+4(E)CX, pm:10+4(E)CX\*1/27+4(E)CX\*2, vm:25+4(E)CX

## Instruction Timing

13+6(E)CX, pm:7+6(E)CX\*1/27+6(E)CX\*2, vm:27+6(E)CX

## Instruction Timing

16+8(E)CX, pm:10+8(E)CX\*1/30+8(E)CX\*2, vm:29+8(E)CX

## Instruction Timing

17+5(E)CX, pm:11+5(E)CX\*1/31+5(E)CX\*2, vm:30+5(E)CX

## Instruction Timing

4+3\*CX

## Instruction Timing

5\*3,13\*4,12+3(E)CX\*5

## Instruction Timing

5\*3,7+4(E)CX\*6

## Instruction Timing

5\*3,7+5(E)CX\*6

## Instruction Timing

5\*3,7+7(E)CX\*6

---

## Instruction Timing

5+12(E)CX, pm:6+5(E)CX\*1/26+5(E)CX\*2, vm:26+5(E)CX

---

## Instruction Timing

5+15\*15

---

## Instruction Timing

5+15\*N

---

## Instruction Timing

5+22\*N

---

## Instruction Timing

5+4(E)CX

---

## Instruction Timing

5+4\*CX

---

## Instruction Timing

5+5(E)CX

---

## Instruction Timing

5+8\*N

---

## Instruction Timing

$5+9*N$

-----

## Instruction Timing

$6*3,13*4$

-----

## Instruction Timing

$6*3,13*4,13(E)CX*5$

-----

## Instruction Timing

$6*3,9(E)CX*6$

-----

## Instruction Timing

$6+11*N$

-----

## Instruction Timing

$6+9*N$

-----

## Instruction Timing

$7*3,7+3(E)CX*6$

-----

## Instruction Timing

$7*3,8+4(E)CX*6$

-----

## Instruction Timing

7\*3,9+4(E)CX\*6

-----

## Instruction Timing

8+8\*N

-----

## Instruction Timing

9+10\*CX

-----

## Instruction Timing

9+15\*N

-----

## Instruction Timing

9+17\*CX

-----

## Instruction Timing

9+22\*N

-----

## Instruction Timing

N/A

-----

## Intel Instruction Set

The following section describes the individual processor instructions in detail.

-----

## Protected Mode Exceptions

None

---

## Real Address Mode Exceptions

None

---

## Virtual 8086 Mode Exceptions

None

---

## Flags Affected

OF DF IF SF ZF AF PF CF

None

---

## FPU Flags Affected

C0 C1 C2 C3

? \* ? ?

C1 as described in [FPU Flags Affected](#); C0, C2, C3 undefined.

---

## FPU Flags Affected

C0 C1 C2 C3

? ? ? ?

C0, C1, C2, C3 undefined.

---

## FPU Flags Affected

C0 C1 C2 C3  
\* \* \* \*

C0, C1, C2, C3 as described in [FPU Flags Affected](#).

## FPU Flags Affected

C0 C1 C2 C3  
\* \* \* \*

C0, C1, C2, C3 as loaded.

## AAA-ASCII Adjust after Addition

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
37	AAA	X	X	X	X	X	X	ASCII adjust AL after addition

## Description

Run the AAA instruction only following an ADD instruction that leaves a byte result in the AL register. The lower nibbles of the operands of the ADD instruction should be in the range 0 through 9 (BCD digits). In this case, the AAA instruction adjusts the AL register to contain the correct decimal digit result. If the addition produced a decimal carry, the AH register is incremented, and the CF and AF flags are set. If this same addition also produced FH in the upper nibble of AL then AH is incremented again. If there was no decimal carry, the CF and AF flags are cleared and the AH register is unchanged. In either case, the AL register is left with its top nibble set to 0. To convert the AL register to an ASCII result, follow the AAA instruction with OR AL, 30H.

## Operation

```
ALcarry ` AL > 0F9H; (* 1 if true *)
IF ((AL AND 0FH) > 9) OR (AF = 1)
THEN
    AL ` (AL + 6) AND 0FH;
    AH ` AH + 1 + ALcarry;
    AF ` 1;
    CF ` 1;
ELSE
    AF ` 0;
    CF ` 0;
    AL ` AL AND 0FH;
FI;
```

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
?			?	?	*	?	*

The AF and CF flags are set if there is a decimal carry, cleared if there is no decimal carry; the OF, SF, ZF, and PF flags are undefined.

## Related Information

- [Description](#)
- [Flags Affected](#)
- [Operation](#)
- [Protected Mode Exceptions](#)
- [Real Address Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

## AAD-ASCII Adjust AX before Division

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D5 0A	AAD	X	X	X	X	X	X	ASCII adjust before division

---

## Description

The AAD instruction is used to prepare two unpacked BCD digits (the least-significant digit in the AL register, the most-significant digit in the AH register) for a division operation that will yield an unpacked result. This is accomplished by setting the AL register to AL+ (second byte of opcode \* AH), and then clearing the AH register. The AX register is then equal to the binary equivalent of the original unpacked two-digit number.

---

## Operation

```
regAL = AL;
regAH = AH;
AL ← (regAH * imm8 + regAL) AND 0FFH;
AH ← 0;
```

**Note:** imm8 has the value of the instruction's second byte. The second byte under normal assembly of this instruction will be 0A, however, explicit modification of this byte will result in the operation described above and may alter results.

---

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
?			*	*	?	*	?

The SF, ZF, and PF flags are set according to the result; the OF, AF, and CF flags are undefined.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## AAM-ASCII Adjust AX after Multiply

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D4 0A	AAM	X	X	X	X	X	X	ASCII adjust AX after

---

## Description

Run the AAM instruction only after running a MUL instruction between two unpacked BCD digits that leaves the result in the AX register. Because the result is less than 100, it is contained entirely in the AL register. The AAM instruction unpacks the AL result by dividing AL by the second byte of the opcode, leaving the quotient (most-significant digit) in the AH register and the remainder (least-significant digit) in the AL register.

---

## Operation

regAL ← AL;  
AH ← regAL / imm8;  
AL ← regAL MOD imm8;

**Note:** imm8 has the value of the instruction's second byte. The second byte under normal assembly of this instruction will be 0A., however, explicit modification of this byte will result in the operation described above and may alter results.

---

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
?		*	*	?	*	?	

The SF, ZF, and PF flags are set according to the result; the OF, AF, and CF flags are undefined.

---

## Related Information

[Description](#)

[Flags Affected](#)

# AAS-ASCII Adjust AL after Subtraction

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
3F	AAS	X	X	X	X	X	X	ASCII adjust AL after subtraction

## Description

Run the AAS instruction only after a SUB instruction that leaves the byte result in the AL register. The lower nibbles of the operands of the SUB instruction must have been in the range of 0 through 9 (BCD digits). In this case, the AAS instruction adjusts the AL register so it contains the correct decimal digit result. If the subtraction produced a decimal carry, the AH register is decremented, and the CF and AF flags are set. If no decimal carry occurred, the CF and AF flags are cleared, and the AH register is unchanged. In either case, the AL register is left with its top nibble set to 0. To convert the AL result to an ASCII result, follow the AAS instruction with OR AL, 30H.

## Operation

```
ALborrow ← AL < 6; (* 1 if true *)
IF (AL AND 0FH) > 9 OR AF = 1
THEN
    AL ← (AL - 6) AND 0FH;
    AH ← AH - 1 - ALborrow;
    AF ← 1;
    CF ← 1;
ELSE
    CF ← 0;
    AF ← 0;
    AL ← AL AND 0FH
FI;
```

## Flags Affected

OF DF IF SF ZF AF PF CF  
?            ?   ?   \*   ?   \*

The AF and CF flags are set if there is a decimal carry, cleared if there is no decimal carry; the OF, SF, ZF, and PF flags are undefined.

# Related Information

- [Description](#)
- [Flags Affected](#)
- [Operation](#)
- [Protected Mode Exceptions](#)
- [Real Address Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

# ADC-Add with Carry

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
14 ib	ADC AL,imm8	X	X	X	X	X	X	Add with carry immediate byte to AL
15 iw	ADC AX,imm16	X	X	X	X	X	X	Add with carry immediate word to AX
15 id	ADC EAX,imm32				X	X	X	Add with carry immediate dword to EAX
80 /2 ib	ADC r/m8,imm8	X	X	X	X	X	X	Add with carry immediate byte to r/m byte
81 /2 iw	ADC r/m16,imm16	X	X	X	X	X	X	Add with carry immediate word to r/m word
81 /2 id	ADC r/m32,imm32				X	X	X	Add with carry immediate dword to r/m dword
83 /2 ib	ADC r/m16,imm8	X	X	X	X	X	X	Add with carry sign-extended immediate byte to r/m word
83 /2 id	ADC r/m32,imm8				X	X	X	Add with carry sign-extended immediate byte into r/m dword
10 /r	ADC r/m8,r8	X	X	X	X	X	X	Add with carry byte register to r/m byte

11	/r	ADC <code>r/m16,r16</code>	X X X X X X	Add with carry word register to r/m word
11	/r	ADC <code>r/m32,r32</code>	X X X	Add with carry dword register to r/m word
12	/r	ADC <code>r8,r/m8</code>	X X X X X X	Add with carry r/m byte to byte register
13	/r	ADC <code>r16,r/m16</code>	X X X X X X	Add with carry r/m word to word register
13	/r	ADC <code>r32,r/m32</code>	X X X	Add with carry r/m dword to dword register

## Description

The ADC instruction performs an integer addition of the two operands DEST and SRC and the carry flag, CF. The result of the addition is assigned to the first operand (DEST), and the flags are set accordingly. The ADC instruction is usually run as part of a multi-byte or multi-word addition operation. When an immediate byte value is added to a word or doubleword operand, the immediate value is first sign-extended to the size of the word or doubleword operand.

## Operation

DEST ← DEST + SRC + CF;

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
*			*	*	*	*	*

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

# Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Related Information

- Description
- Flags Affected
- Operation
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

## ADD-Add

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
04 ib	ADD AL,imm8	X	X	X	X	X	X	Add immediate byte to AL
05 iw	ADD AX,imm16	X	X	X	X	X	X	Add immediate word to AX
05 id	ADD EAX,imm32				X	X	X	Add immediate dword to EAX
80 /0 ib	ADD r/m8,imm8	X	X	X	X	X	X	Add immediate byte to r/m byte
81 /0 iw	ADD r/m16,imm16	X	X	X	X	X	X	Add immediate word to r/m word
81 /0 id	ADD r/m32,imm32				X	X	X	Add immediate dword to r/m dword
83 /0 ib	ADD r/m16,imm8	X	X	X	X	X	X	Add sign-extended immediate byte to r/m word
83 /0 id	ADD r/m32,imm8				X	X	X	Add sign-extended immediate byte to r/m dword
00 /r	ADD r/m8,r8	X	X	X	X	X	X	Add byte register to r/m byte
01 /r	ADD r/m16,r16	X	X	X	X	X	X	Add word register to r/m word
01 /r	ADD r/m32,r32				X	X	X	Add dword register to r/m dword

02	/r	ADD r8,r/m8	X X X X X X	Add r/m byte to byte register
03	/r	ADD r16,r/m16	X X X X X X	Add r/m word to word register
03	/r	ADD r32,r/m32	X X X	Add r/m dword to dword register

## Description

The ADD instruction performs an integer addition of the two operands (DEST and SRC). The result of the addition is assigned to the first operand (DEST), and the flags are set accordingly.

When an immediate byte is added to a word or doubleword operand, the immediate value is sign-extended to the size of the word or doubleword operand.

## Operation

DEST ← DEST + SRC;

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
*			*	*	*	*	*

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

## Protected Mode Exceptions

#GP(0) is the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

# Related Information

- [Description](#)
- [Flags Affected](#)
- [Operation](#)
- [Protected Mode Exceptions](#)
- [Real Address Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

# AND-Logical AND

# Details Table

Encoding	Instruction	0 1 2 3 4 5	Description
24 ib	AND AL,imm8	X X X X X X	AND immediate byte to AL
25 iw	AND AX,imm16	X X X X X X	AND immediate word to AX
25 id	AND EAX,imm32	X X X	AND immediate dword to EAX
80 /r ib	AND r/m8,imm8	X X X X X X	AND immediate byte to r/m byte
81 /4 iw	AND r/m16,imm16	X X X X X X	AND immediate word to r/m word
81 /r id	AND r/m32,imm32	X X X	AND immediate dword to r/m dword
83 /4 ib	AND r/m16,imm8	X X X	AND sign-extended immediate byte to r/m word
83 /4 ib	AND r/m32,imm8	X X X	AND sign-extended immediate byte with r/m dword
20 /r	AND r/m8,r8	X X X X X X	AND byte register to r/m byte
21 /r	AND r/m16,r16	X X X X X X	AND word register to r/m word
21 /r	AND r/m32,r32	X X X	AND dword register to r/m dword
22 /r	AND r8,r/m8	X X X X X X	AND r/m byte to byte register
23 /r	AND r16,r/m16	X X X X X X	AND r/m word to word register
23 /r	AND r32,r/m32	X X X	AND r/m dword to dword register

---

## Description

Each bit of the result of the AND instruction is a 1 if both corresponding bits of the operands are 1; otherwise, it becomes a 0.

---

## Operation

DEST ← DEST AND SRC;  
CF ← 0;  
OF ← 0;

---

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
0			*	*	?	*	0

---

The CF and OF flags are cleared; the PF, SF, and ZF flags are set according to the result; the AF flag is undefined.

---

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

# ARPL-Adjust RPL Field of Selector

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
63 /r	ARPL <i>r/m16,r16</i>			X	X	X	X	Adjust RPL of <i>r/m16</i> to not less than RPL of <i>r16</i>

---

## Description

The ARPL instruction has two operands. The first operand is a 16-bit memory variable or word register that contains the value of a selector. The second operand is a word register. If the RPL field ("requested privilege level"-bottom two bits) of the first operand is less than the RPL field of the second operand, the ZF flag is set and the RPL field of the first operand is increased to match the second operand. Otherwise, the ZF flag is cleared and no change is made to the first operand.

The ARPL instruction appears in operating system software, not in application programs. It is used to guarantee that a selector parameter to a subroutine does not request more privilege than the caller is allowed. The second operand of the ARPL instruction is normally a register that contains the CS selector value of the caller.

---

## Operation

```
IF RPL bits(0,1) of DEST < RPL bits(0,1) of SRC
THEN
    ZF ` 1;
    RPL bits(0,1) of DEST ` RPL bits (0,1) of SRC;
ELSE
    ZF ` 0;
FI;
```

---

# Flags Affected

OF DF IF SF ZF AF PF CF

\*

The ZF flag is set if the RPL field of the first operand is less than that of the second operand, otherwise ZF is cleared.

## Protected Mode Exceptions

#GP(0) if the result is a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 6; the ARPL instruction is not recognized in Real Address Mode.

## Virtual 8086 Mode Exceptions

Interrupt 6; the ARPL instruction is not recognized in Virtual 8086 Mode.

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

## BOUND-Check Array Index Against Bounds

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
62 /r	BOUND r16,m16&16	X	X	X	X	X	X	Check if r16 is within m16&16 bounds (passes test)
62 /r	BOUND r32,m32&32	X	X	X	X	X	X	Check if r32 is within m32&32 bounds (passes test)

## Description

The BOUND instruction ensures that a signed array index is within the limits specified by a block of memory consisting of an upper and a lower bound. Each bound uses one word when the operand-size attribute is 16-bits and a doubleword when the operand-size attribute is 32-bits. The first operand (a register) must be greater than or equal to the first bound in memory (lower bound), and less than or equal to the second bound in memory (upper bound) plus the number of bytes occupied for the operand size. If the register is not within bounds, an Interrupt 5 occurs; the return EIP points to the BOUND instruction.

The bounds limit data structure is usually placed just before the array itself, making the limits addressable via a constant offset from the beginning of the array.

## Operation

IF (LeftSRC < [RightSRC] OR LeftSRC > [RightSRC +  
OperandSize/8])  
(\* Under lower bound or over upper bound \*)  
THEN Interrupt 5;  
FI;

## Protected Mode Exceptions

Interrupt 5 if the bounds test fails, as described above; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

The second operand must be a memory operand, not a register. If the BOUND instruction is run with a ModR/M byte representing a register as the second operand, #UD occurs.

## Real Address Mode Exceptions

Interrupt 5 if the bounds test fails; Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; Interrupt 6 if the second operand is a register.

# Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

## BSF-Bit Scan Forward

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F BC	BSF <a href="#">r16</a> , <a href="#">r/m16</a>				X	X	X	Bit scan forward on r/m word
0F BC	BSF <a href="#">r32</a> , <a href="#">r/m32</a>				X	X	X	Bit scan forward on r/m dword

## Description

The BSF instruction scans the bits in the second word or doubleword operand starting with bit 0. The ZF flag is set if all the bits are 0; otherwise, the ZF flag is cleared and the destination register is loaded with the bit index of the first set bit.

## Operation

IF *r/m* = 0  
THEN

```

ZF ` 1;
register ` UNDEFINED;
ELSE
temp ` 0;
ZF ` 0;
WHILE BIT[m,temp] = 0
DO
temp ` temp + 1;
register ` temp;
OD;
FI;

```

-----

## Flags Affected

```

OF DF IF SF ZF AF PF CF
?      ? * ? ? ?

```

The ZF flag is set if all bits are 0; otherwise, the ZF flag is cleared. OF, SF, AF, PF, CF = undefined.

-----

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

-----

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

# BSR-Bit Scan Reverse

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F BD	BSR <span>r16,r/m16</span>				X	X	X	Bit scan reverse on r/m word
0F BD	BSR <span>r32,r/m32</span>				X	X	X	Bit scan reverse on r/m dword

## Description

The BSR instruction scans the bits in the second word or doubleword operand from the most significant bit to the least significant bit. The ZF flag is set if all the bits are 0; otherwise, the ZF flag is cleared and the destination register is loaded with the bit index of the first set bit found when scanning in the reverse direction.

## Operation

```
IF r/m = 0
THEN
    ZF ` 1;
    register ` UNDEFINED;
ELSE
    temp ` OperandSize =1;
    ZF ` 0;
    WHILE BIT[r/m,temp] = 0
    DO
        temp ` temp - 1;
        register ` temp;
    OD;
FI;
```

## Flags Affected

OF DF IF SF ZF AF PF CF  
?        ?   \*   ?   ?   ?

The ZF flag is set if all bits are 0; otherwise, the ZF flag is cleared. OS, SF, AF, PF, CF = undefined.

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Related Information

- [Description](#)
- [Flags Affected](#)
- [Operation](#)
- [Protected Mode Exceptions](#)
- [Real Address Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

## BSWAP-Byte Swap

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F C8+rd	BSWAP <a href="#">r32</a>					<a href="#">X</a>	<a href="#">X</a>	Swap bytes to convert little/big endian data in a 32-bit register to big/little endian form

## Description

The BSWAP instruction reverses the byte order of a 32-bit register, converting a value in little/big endian form to big/little endian form. When BSWAP is used with 16-bit operand size, the result left in the destination register is undefined.

## Operation

TEMP  $\leftarrow$  r32  
r32(7..0)  $\leftarrow$  TEMP(31..24)  
r32(15..8)  $\leftarrow$  TEMP(23..16)  
r32(23..16)  $\leftarrow$  TEMP(15..8)  
r32(31..24)  $\leftarrow$  TEMP(7..0)

## Notes

BSWAP is not supported on Intel386 processors. Include functionally-equivalent code for Intel386 CPUs.

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

## BT-Bit Test

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F A3	BT <code>r/m16,r16</code>				X	X	X	Save bit in carry flag
0F A3	BT <code>r/m32,r32</code>				X	X	X	Save bit in carry flag
0F BA /4 ib	BT <code>r/m16,imm8</code>				X	X	X	Save bit in carry flag
0F BA /4 ib	BT <code>r/m32,imm8</code>				X	X	X	Save bit in carry flag

## Description

The BT instruction saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the CF flag.

## Operation

CF ← BIT [LeftSRC, RightSRC];

## Flags Affected

OF DF IF SF ZF AF PF CF  
\*

The CF flag contains the value of the selected bit.

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

The index of the selected bit can be given by the immediate constant in the instruction or by a value in a general register. Only an 8-bit immediate value is used in the instruction. This operand is taken modulo 32, so the range of immediate bit offsets is 0..31. This allows any bit within a register to be selected. For memory bit strings, this immediate field gives only the bit offset within a word or doubleword.

Immediate bit offsets larger than 31 are supported by some assemblers by using the immediate bit offset field in combination with the displacement field of the memory operand. In this case, the low-order 3 to 5 bits (3 for 16 bit operands, 5 for 32-bit operands) of the immediate bit offset are stored in the immediate bit offset field, and the high-order bits are shifted and combined with the byte displacement in the addressing mode by the assembler. The processor will ignore the high-order bits if they are not zero.

When accessing a bit in memory, the processor may access four bytes starting from the memory address given by:  
Effective Address + (4 \* (BitOffset DIV 32))

for a 32-bit operand size, or two bytes starting from the memory address given by:  
Effective Address + (2 \* (BitOffset DIV 16))

for a 16-bit operand size. It may do so even when only a single byte needs to be accessed in order to reach the given bit. You must therefore avoid referencing areas of memory close to address space holes. In particular, avoid references to memory-mapped I/O registers. Instead, use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## BTC-Bit Test and Complement

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F BB	BTC <code>r/m16,r16</code>				X	X	X	Save bit in carry flag and complement
0F BB	BTC <code>r/m32,r32</code>				X	X	X	Save bit in carry flag and complement
0F BA /7 ib	BTC <code>r/m16,imm8</code>				X	X	X	Save bit in carry flag and complement
0F BA /7 ib	BTC <code>r/m32,imm8</code>				X	X	X	Save bit in carry flag and complement

## Description

The BTC instruction saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the CF flag and then complements the bit.

## Operation

CF  $\leftarrow$  BIT[LeftSRC, RightSRC];  
 BIT[LeftSRC, RightSRC]  $\leftarrow$  NOT BIT[LeftSrc, RightSRC];

## Flags Affected

OF DF IF SF ZF AF PF CF  
 \*

The CF flag contains the complement of the selected bit.

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, and GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

The index of the selected bit can be given by the immediate constant in the instruction or by a value in a general register. Only an 8-bit immediate value may be used in the instruction. This operand is taken modulo 32, so the range of immediate bit offsets is 0..31. This allows any bit within a register to be selected. For memory bit strings, this immediate field gives only the bit offset within a word or doubleword.

Immediate bit offsets larger than 31 are supported by some assemblers by using the immediate bit offset field in combination with the displacement field of the memory operand. In this case, the low-order 3 to 5 bits (3 for 16-bit operands, 5 for 32-bit operands) of the immediate bit offset are stored in the immediate bit offset field, and the high-order bits are shifted and combined with the byte displacement in the addressing mode by the assembler. The processor will ignore the high order bits if they are not zero.

When accessing a bit in memory, the processor may access four bytes starting from the memory address given by:  
Effective Address + (4 \* (BitOffset DIV 32))

for a 32-bit operand size, or two bytes starting from the memory address given by:  
Effective Address + (2 \* (BitOffset DIV 16))

for a 16-bit operand size. It may do so even when only a single byte needs to be accessed in order to reach the given bit. Therefore, referencing areas of memory close to address space holes should be avoided. In particular, avoid references to memory-mapped I/O registers. Instead, use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## BTR-Bit Test and Reset

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F B3	BTR <a href="#">r/m16,r16</a>					X	X	Save bit in carry flag and reset
0F B3	BTR <a href="#">r/m32,r32</a>					X	X	Save bit in carry flag and reset
0F BA /6 ib	BTR <a href="#">r/m16,imm8</a>					X	X	Save bit in carry flag and reset
0F BA /6 ib	BTR <a href="#">r/m32,imm8</a>					X	X	Save bit in carry flag and reset

## Description

The BTR instruction saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the CF flag and then stores 0 in the bit.

## Operation

```
CF ← BIT[LeftSRC, RightSRC];
BIT[LeftSRC, RightSRC] ← 0;
```

## Flags Affected

OF DF IF SF ZF AF PF CF

\*

The CF flag contains the value of the selected bit.

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

The index of the selected bit can be given by the immediate constant in the instruction or by a value in a general register. Only an 8-bit immediate value is used in the instruction. This operand is taken modulo 32, so the range of immediate bit offsets is 0..31. This allows any bit within a register to be selected. For memory bit strings, this immediate field gives only the bit offset within a word or doubleword.

Immediate bit offsets larger than 31 are supported by some assemblers by using the immediate bit offset field in combination with the displacement field of the memory operand. In this case, the low-order 3 to 5-bits (3 for 16-bit operands, 5 for 32-bit operands) of the immediate bit offset are stored in the immediate bit offset field, and the high-order bits are shifted and combined with the byte displacement in the addressing mode by the assembler. The processor will ignore the high-order bits if they are not zero.

When accessing a bit in memory, the processor may access four bytes starting from the memory address given by:  
Effective Address + 4 \* (BitOffset DIV 32)

for a 32-bit operand size, or two bytes starting from the memory address given by:  
Effective Address + 2 \* (BitOffset DIV 16)

for a 16-bit operand size. It may do so even when only a single byte needs to be accessed in order to reach the given bit. You must therefore avoid referencing areas of memory close to address space holes. In particular, avoid references to memory-mapped I/O registers. Instead, use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## BTS-Bit Test and Set

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F AB	BTS <code>r/m16,r16</code>				X	X	X	Save bit in carry flag and set
0F AB	BTS <code>r/m32,r32</code>				X	X	X	Save bit in carry flag and set
0F BA /5 ib	BTS <code>r/m16,imm8</code>				X	X	X	Save bit in carry flag and set
0F BA /5 ib	BTS <code>r/m32,imm8</code>				X	X	X	Save bit in carry flag and set

## Description

The BTS instruction saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the CF flag and then stores 1 in the bit.

## Operation

```
CF ← BIT[LeftSRC, RightSRC];
BIT[LeftSRC, RightSRC] ← 1;
```

## Flags Affected

OF DF IF SF ZF AF PF CF

\*

The CF flag contains the value of the selected bit.

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

The index of the selected bit can be given by the immediate constant in the instruction or by a value in a general register. Only an 8-bit immediate value is used in the instruction. This operand is taken modulo 32, so the range of immediate bit offsets is 0..31. This allows any bit within a register to be selected. For memory bit strings, this immediate field gives only the bit offset within a word or doubleword.

Immediate bit offsets larger than 31 are supported by some assemblers by using the immediate bit offset field in combination with the displacement field of the memory operand. In this case, the low-order 3 to 5 bits (3 for 16-bit operands, 5 for 32-bit operands) of the immediate bit offset are stored in the immediate bit offset field, and the high-order bits are shifted and combined with the byte displacement in the addressing mode by the assembler. The processor will ignore the high-order bits if they are not zero.

When accessing a bit in memory, the processor may access four bytes starting from the memory address given by:  
Effective Address + (4 \* (BitOffset DIV 32))

for a 32-bit operand size, or two bytes starting from the memory address given by:  
Effective Address + (2 \* (BitOffset DIV 16))

for a 16-bit operand size. It may do this even when only a single byte needs to be accessed in order to get at the given bit. You must therefore be careful to avoid referencing areas of memory close to address space holes. In particular, avoid references to memory-mapped I/O registers. Instead, use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## CALL-Call Procedure

---

## Details Table

Encoding	Instruction	0 1 2 3 4 5	Description
E8 cw	CALL rel16	X X X X X X	Call near, displacement relative to next instruction
FF /2	CALL r/m16	X X X X X X	Call near, register indirect/memory indirect
9A cd	CALL ptr16:16	X X X X X X	Call intersegment, to full pointer given
9A cd	CALL ptr16:16	X X X X	Call gate, same privilege
9A cd	CALL ptr16:16	X X X X	Call gate, more privilege, no parameters
9A cd	CALL ptr16:16	X X X X	Call gate, more privilege, x parameters
9A cd	CALL ptr16:16	X X X X	Call to task (via task state segment/task gate for 286)
FF /3	CALL m16:16	X X X X X X	Call intersegment, address at r/m dword
FF /3	CALL m16:16	X X X X	Call gate, same privilege
FF /3	CALL m16:16	X X X X	Call gate, more privilege, no parameters
FF /3	CALL m16:16	X X X X	Call gate, more privilege, x parameters
FF /3	CALL m16:16	X X X X	Call to task (via task state segment/task gate for 286)
E8 cd	CALL rel32	X X X	Call near, displacement relative to next instruction
FF /2	CALL r/m32	X X X	Call near, register indirect/memory indirect
9A cp	CALL ptr16:32	X X X	Call intersegment, to full pointer given
9A cp	CALL ptr16:32	X X X	Call gate, same privilege
9A cp	CALL ptr16:32	X X X	Call gate, more privilege, no parameters
9A cp	CALL ptr16:32	X X X	Call gate, more privilege, x parameters
9A cp	CALL ptr16:32	X X X	Call to task
FF /3	CALL m16:32	X X X	Call intersegment, address at r/m fword
FF /3	CALL m16:32	X X X	Call gate, same privilege
FF /3	CALL m16:32	X X X	Call gate, more privilege, no parameters
FF /3	CALL m16:32	X X X	Call gate, more privilege, x parameters
FF /3	CALL m16:32	X X X	Call to task

## Description

The CALL instruction causes the procedure named in the operand to be run. When the procedure is complete (a return instruction is run within the procedure), processing continues at the instruction that follows the CALL instruction.

The action of the different forms of the instruction are described below.

Near calls are those with destination of type *r/m16*, *r/m32*, *rel16*, *rel32*; changing or saving the segment register value is not necessary. The CALL *rel16* and CALL *rel32* forms add a signed offset to the address of the instruction following the CALL instruction to determine the destination. The *rel16* form is used when the instruction's operand-size attribute is 16-bits; *rel32* is used when the operand-size attribute is 32-bits. The result is stored in the 32-bit EIP register. With *rel16*, the upper 16-bits of the EIP register are cleared, resulting in an offset whose value does not exceed 16-bits. CALL *r/m16* and CALL *r/m32* specify a register or memory location from which the absolute segment offset is fetched. The offset fetched from *r/m* is 32-bits for an operand-size attribute of 32 (*r/m32*), or 16-bits for an operand-size of 16 (*r/m16*). The offset of the instruction following the CALL instruction is pushed onto the stack. It will be popped by a near RET instruction within the procedure. The CS register is not changed by this form of CALL.

The far calls, CALL *ptr:16* and CALL *ptr16:32*, use a four-byte or six-byte operand as a long pointer to the procedure called. The CALL *m16:16* and *m16:32* forms fetch the long pointer from the memory location specified (indirection). In Real Address Mode or Virtual 8086 Mode, the long pointer provides 16-bits for the CS register and 16 or 32-bits for the EIP register (depending on the operand-size attribute). These forms of the instruction push both the CS and IP or EIP registers as a return address.

In Protected Mode, both long pointer forms consult the AR byte in the descriptor indexed by the selector part of the long pointer. Depending on the value of the AR byte, the call will perform one of the following types of control transfers:

- A far call to the same protection level
- An inter-protection level far call
- A task switch

A CALL-indirect-thru-memory, which uses the stack pointer (ESP) as a base register, references memory before the CALL. The base used is the value of the ESP before the instruction runs.

For more information on Protected Mode control transfers, refer to the Intel documentation.

## Operation

```
IF rel16 or rel32 type of call
THEN (* near relative call *)
  IF OperandSize = 16
  THEN
    Push(IP);
    EIP ← (EIP + rel16) AND 0000FFFFH;
  ELSE (* OperandSize = 32 *)
    Push(EIP);
    EIP ← EIP + rel32;
  FI;
FI;

IF r/m16 or r/m32 type of call
THEN (* near absolute call *)
  IF OperandSize = 16
  THEN
    Push(IP);
    EIP ← [r/m16] AND 0000FFFFH;
  ELSE (*OperandSize = 32 *)
    Push(EIP);
    EIP ← [r/m32];
  FI;
FI;

IF (PE = 0 OR (PE = 1 AND VM = 1))
(* real mode or virtual 8086 mode *)
AND instruction = far CALL
(* i.e., operand type is m16:16, m16:32, ptr16:16, ptr16:32 *)
THEN
  IF OperandSize = 16
  THEN
    Push(CS);
    Push(IP); (* address of next instruction; 16 bits *)
  ELSE
```

```

    Push(CS); (* padded with 16 high-order bits *)
    Push(EIP); (* address of next instruction; 32 bits *)
FI;
IF operand type is m16:16 or m16:32
THEN (* indirect far call *)
    IF OperandSize = 16
    THEN
        CS:IP ` [m16:16];
        EIP ` EIP AND 0000FFFFH; (* clear upper 16 bits *)
    ELSE (* OperandSize = 32 *)
        CS:EIP ` [m16:32];
    FI;
FI;
IF operand type is ptr:16 or ptr16:32
THEN (* direct far call *)
    IF OperandSize = 16
    THEN
        CS:IP ` ptr:16;
        EIP ` EIP AND 0000FFFFH; (* clear upper 16 bits *)
    ELSE (* OperandSize = 32 *)
        CS:EIP ` ptr16:32;
    FI;
FI;
FI;

IF (PE = 1 AND VM = 0) (* Protected mode, not V86 mode *)
AND instruction = far CALL
THEN
    If indirect, then check access of EA doubleword;
    #GP(0) if limit violation;
    New CS selector must not be null else #GP(0);
    Check that new CS selector index is within its
        descriptor table limits; else #GP(new CS selector);
    Examine AR byte of selected descriptor for various legal values;
    depending on value:
        go to CONFORMING-CODE-SEGMENT;
        go to NONCONFORMING-CODE-SEGMENT;
        go to CALL-GATE;
        go to TASK-GATE;
        go to TASK-STATE-SEGMENT;
    ELSE #GP(code segment selector);
FI;

CONFORMING-CODE-SEGMENT:
    DPL must be ¼ CPL ELSE #GP(code segment selector);
    Segment must be present ELSE #NP(code segment selector);
    Stack must be big enough for return address ELSE #SS(0);
    Instruction pointer must be in code segment limit ELSE #GP(0);
    Load code segment descriptor into CS register;
    Load CS with new code segment selector;
    Load EIP with zero-extend(new offset);
    IF OperandSize=16 THEN EIP ` EIP AND 0000FFFFH; FI;

NONCONFORMING-CODE-SEGMENT:
    RPL must be ¼ CPL ELSE #GP(code segment selector)
    DPL must be = CPL ELSE #GP(code segment selector)
    Segment must be present ELSE #NP(code segment selector)
    Stack must be big enough for return address ELSE #SS(0)
    Instruction pointer must be in code segment limit ELSE #GP(0)
    Load code segment descriptor into CS register
    Load CS with new code segment selector
    Set RPL of CS to CPL
    Load EIP with zero-extend(new offset);
    IF OperandSize=16 THEN EIP ` EIP AND 0000FFFFH; FI;

CALL-GATE
    Call gate DPL must be ¼ CPL ELSE #GP(call gate selector)
    Call gate DPL must be ¼ RPL ELSE #GP(call gate selector)
    Call gate must be present ELSE #NP(call gate selector)
    Examine code segment selector in call gate descriptor:
        Selector must not be null ELSE #GP(0)
        Selector must be within its descriptor table
            limits ELSE #GP(code segment selector)
    AR byte of selected descriptor must indicate code
        segment ELSE #GP(code segment selector)
    DPL of selected descriptor must be ¼ CPL ELSE

```

```
#GP(code segment selector)
IF non-conforming code segment AND DPL < CPL
THEN go to MORE-PRIVILEGE
ELSE go to SAME-PRIVILEGE
FI;
```

#### MORE-PRIVILEGE:

```
Get new SS selector for new privilege level from TSS
Check selector and descriptor for new SS:
  Selector must not be null ELSE #TS(0)
  Selector index must be within its descriptor
  table limits ELSE #TS(SS selector)
  Selector's RPL must equal DPL of code segment
  ELSE #TS(SS selector)
  Stack segment DPL must equal DPL of code
  segment ELSE #TS(SS selector)
  Descriptor must indicate writable data segment
  ELSE #TS(SS selector)
  Segment present ELSE #SS(SS selector)
IF OperandSize=32
THEN
  New stack must have room for parameters plus 16 bytes
  ELSE #SS(SS selector)
  EIP must be in code segment limit ELSE #GP(0)
  Load new SS:eSP value from TSS
  Load new CS:EIP value from gate
ELSE
  New stack must have room for parameters plus 8 bytes
  ELSE #SS(SS selector)
  IP must be in code segment limit ELSE #GP(0)
  Load new SS:eSP value from TSS
  Load new CS:IP value from gate
FI;
Load CS descriptor
Load SS descriptor
Push long pointer of old stack onto new stack
Get word count from call gate, mask to 5 bits
Copy parameters from old stack onto new stack
Push return address onto new stack
Set CPL to stack segment DPL
Set RPL of CS to CPL
```

#### SAME-PRIVILEGE:

```
IF OperandSize=32
THEN
  Stack must have room for 6-byte return address (padded to 8 bytes)
  ELSE #SS(0)
  EIP must be within code segment limit ELSE #GP(0)
  Load CS:EIP from gate
ELSE
  Stack must have room for 4-byte return address ELSE #SS(0)
  IP must be within code segment limit ELSE #GP(0)
  Load CS:IP from gate
FI;
Push return address onto stack
Load code segment descriptor into CS register
Set RPL of CS to CPL
```

#### TASK-GATE:

```
Task gate DPL must be  $\frac{3}{4}$  CPL ELSE #TS(gate selector)
Task gate DPL must be  $\frac{3}{4}$  RPL ELSE #TS(gate selector)
Task Gate must be present ELSE #NP(gate selector)
Examine selector to TSS, given in Task Gate descriptor:
  Must specify global in the local/global bit ELSE #TS(TSS selector)
  Index must be within GDT limits ELSE #TS(TSS selector)
  TSS descriptor AR byte must specify nonbusy TSS
  ELSE #TS(Tss selector)
  Task State Segment must be present ELSE #NP(TSS selector)
SWITCH-TASKS (with nesting) to TSS
IP must be in code segment limit ELSE #TS(0)
```

#### TASK-STATE-SEGMENT

```
TSS DPL must be  $\frac{3}{4}$  CPL ELSE #TS(TSS selector)
TSS DPL must be  $\frac{3}{4}$  RPL ELSE #TS(TSS selector)
TSS descriptor AR byte must specify available TSS
ELSE #TS(TSS selector)
```

Task State Segment must be present ELSE #NP(TSS selector)  
SWITCH-TASKS (with nesting) to TSS  
IP must be in code segment limit ELSE #TS(0)

---

## Flags Affected

OF DF IF SF ZF AF PF CF

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

---

## Protected Mode Exceptions

For far calls: #GP, #NP, #SS, and #TS, as indicated in the "Operation" section.

For near direct calls: #GP(0) if procedure location is beyond the code segment limits; #SS(0) if pushing the return address exceeds the bounds of the stack segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

For a near indirect call: #GP(0) for an illegal memory operand effective address is the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #GP(0) if the indirect offset obtained is beyond the code segment limits; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

Any far call from a 32-bit code segment to a 16-bit code segment should be made from the first 64 Kbytes of the 32-bit code segment, because the operand-size attribute of the instruction is set to 16, allowing only a 16-bit return address offset to be saved.

---

## Related Information

[Description](#)

- [Flags Affected](#)
- [Notes](#)
- [Operation](#)
- [Protected Mode Exceptions](#)
- [Real Address Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

---

# CBW/CWDE-Convert Byte to Word/Convert Word to Doubleword

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
98	CBW	X	X	X	X	X	X	AX ` sign extend of AL
98	CWDE			X	X	X		EAX ` sign-extend of AX

---

## Description

The CBW instruction converts the signed byte in the AL register to a signed word in the AX register by extending the most significant bit of the AL register (the sign bit) into all of the bits of the AH register. The CWDE instruction converts the signed word in the AX register to a doubleword in the EAX register by extending the most significant bit of the AX register into the two most significant bytes of the EAX register. Note that the CWDE instruction is different from the CWD instruction. The CWD instruction uses the DX:AX register pair rather than the EAX register as a destination.

---

## Operation

```
IF OperandSize = 16 (* instruction = CBW *)
THEN AX ` Sign Extend(AL);
ELSE (* OperandSize = 32, instruction = CWDE *)
  EAX ` Sign Extend(AX);
FI;
```

---

## Related Information

- [Description](#)

[Flags Affected](#)  
[Operation](#)  
[Protected Mode Exceptions](#)  
[Real Address Mode Exceptions](#)  
[Virtual 8086 Mode Exceptions](#)

---

# CDQ-Convert Double to Quad

See entry for CWD/CDQ-Convert Word to Double/Convert Double to Quad.

---

# CLC-Clear Carry Flag

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
F8	CLC	X	X	X	X	X	X	Clear carry flag

---

## Description

The CLC instruction clears the CF flag. It does not affect other flags or registers.

---

## Operation

CF ← 0;

---

## Flags Affected

OF DF IF SF ZF AF PF CF  
0

The CF flag is cleared.

## Related Information

- Description
- Flags Affected
- Operation
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

## CLD-Clear Direction Flag

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
FC	CLD	X	X	X	X	X	X	Clear direction flag

## Description

The CLD instruction clears the direction flag. No other flags or registers are affected. After a CLD instruction is run, string operations will increment the index registers (SI and/or DI) that they use.

## Operation

DF ← 0;

# Flags Affected

OF DF IF SF ZF AF PF CF  
0

The DF flag is cleared.

## Related Information

- Description
- Flags Affected
- Operation
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

## CLI-Clear Interrupt Flag

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
FA	CLI	X	X	X	X	X	X	Clear interrupt flag; interrupts disabled when interrupt flag cleared

## Description

The CLI instruction clears the IF flag if the current privilege level is at least as privileged as IOPL. No other flags are affected. External interrupts are not recognized at the end of the CLI instruction from that point on until the IF flag is set.

# Operation

```
IF PE = 0
THEN
  IF `0;
ELSE
  IF VM = 0 (* Running in protected Mode *)
  THEN
    IF IOPL = 3
    THEN IF ` 0;
    ELSE IF CPL  $\geq$  IOPL
    THEN IF ` 0;
    ELSE #GP(0);
    FI;
  FI;
ELSE (* Running in Virtual-8086 mode *)
  IF IOPL = 3
  THEN IF `
  ELSE #GP(0);
  FI;
FI;
FI;
```

---

## Decision Table

The following decision table indicates which action in the lower portion of the table is taken given the conditions in the upper portion of the table.

PE =	0	1	1	1	1
VM =	-	0	-	0	1
CPL	-	$\geq$ IOPL	-	>IOPL	-
IOPL	-	-	= 3	-	< 3
IF ` 0	Y	Y	Y		
#GP (0)				Y	Y

### Notes:

-	Don't care
Blank	Action Not Taken
Y	Action in Column 1 taken

---

## Flags Affected

OF DF IF SF ZF AF PF CF

0

IF cleared.

---

# Protected Mode Exceptions

#GP(0) if the current privilege level is greater (has less privilege) than the I/O privilege level in the flags register. The I/O privilege level specifies the least privileged level at which I/O can be performed.

---

# Virtual 8086 Mode Exceptions

#GP(0) as for protected mode.

---

# Related Information

- [Decision Table](#)
- [Description](#)
- [Flags Affected](#)
- [Operation](#)
- [Protected Mode Exceptions](#)
- [Real Address Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

---

# CLTS-Clear Task-Switched Flag in CR0

---

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F 06	CLTS			X	X	X	X	Clear task-switched flag

---

# Description

The CLTS instruction clears the task-switched (TS) flag in the CR0 register. This flag is set by the processor every time a task switch occurs. The TS flag is used to manage processor extensions as follows:

- Every processing of an ESC instruction is trapped if the TS flag is set.
- Processing of a WAIT instruction is trapped if the MP flag and the TS flag are both set.

Thus, if a task switch was made after an ESC instruction was begun, the floating-point unit's context may need to be saved before a new ESC instruction can be issued. The fault handler saves the context and clears the TS flag.

The CLTS instruction appears in operating system software, not in application programs. It is a privileged instruction that can only be run at privilege level 0.

---

## Operation

TS Flag in CR0 ` 0;

---

## Flags Affected

OF DF IF SF ZF AF PF CF

The TS flag is cleared (the TS flag is in the CR0 register, not the flags register).

---

## Protected Mode Exceptions

#GP(0) if the CLTS instruction is run with a current privilege level other than 0.

---

## Real Address Mode Exceptions

None (valid in Real Address Mode to allow initialization for Protected Mode).

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Protected Mode.

---

## Related Information

[Description](#)  
[Flags Affected](#)  
[Operation](#)  
[Protected Mode Exceptions](#)  
[Real Address Mode Exceptions](#)  
[Virtual 8086 Mode Exceptions](#)

---

# CMC-Complement Carry Flag

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
F5	CMC	X	X	X	X	X	X	Complement carry flag

---

## Description

The CMC instruction reverses the setting of the CF flag. No other flags are affected.

---

## Operation

CF ← NOT CF;

---

## Flags Affected

OF DF IF SF ZF AF PF CF  
\*

The CF flag contains the complement of its original value.

# Related Information

- Description
- Flags Affected
- Operation
- Protected Mode Exceptions
- Protected Mode Exceptions
- Virtual 8086 Mode Exceptions

# CMP-Compare Two Operands

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
3C ib	CMP AL,imm8	X	X	X	X	X	X	Compare immediate byte to AL
3D iw	CMP AX,imm16	X	X	X	X	X	X	Compare immediate word to AX
3D id	CMP EAX,imm32				X	X	X	Compare immediate dword to EAX
80 /7 ib	CMP r/m8,imm8	X	X	X	X	X	X	Compare immediate byte to r/m byte
81 /7 iw	CMP r/m16,imm16	X	X	X	X	X	X	Compare immediate word to r/m word
81 /7 id	CMP r/m32,imm32				X	X	X	Compare immediate dword to r/m dword
83 /7 ib	CMP r/m16,imm8	X	X	X	X	X	X	Compare sign-extended immediate byte to r/m word
83 /7 ib	CMP r/m32,imm8				X	X	X	Compare sign-extended immediate byte to r/m dword
38 /r	CMP r/m8,r8	X	X	X	X	X	X	Compare byte register to r/m byte
39 /r	CMP r/m16,r16	X	X	X	X	X	X	Compare word register to r/m word
39 /r	CMP r/m32,r32				X	X	X	Compare dword register to r/m dword
3A /r	CMP r8,r/m8	X	X	X	X	X	X	Compare r/m byte to byte register
3B /r	CMP r16,r/m16	X	X	X	X	X	X	Compare r/m word to word register
3B /r	CMP r32,r/m32				X	X	X	Compare r/m dword to dword register

---

## Description

The CMP instruction subtracts the second operand from the first but, unlike the SUB instruction, does not store the result; only the flags are changed. The CMP instruction is typically used in conjunction with conditional jumps and the SETcc instruction. (Refer to Appendix D for the list of signed and unsigned flag tests provided.) If an operand greater than one byte is compared to an immediate byte, the byte value is first sign-extended.

---

## Operation

LeftSRC - SignExtend(RightSRC);  
(\* CMP does not store a result; its purpose is to set the flags \*)

---

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
*			*	*	*	*	*

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

---

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

# CMPS/CMPSB/CMPSW/CMPSD-Compare String Operands

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
A6	CMPS <a href="#">m8,m8</a>	X	X	X	X	X	X	Compare bytes ES:[(E)DI] (second operand) with [(E)SI] (first operand)
A7	CMPS <a href="#">m16,m16</a>	X	X	X	X	X	X	Compare words ES:[(E)DI] (second operand) with [(E)SI] (first operand)
A7	CMPS <a href="#">m32,m32</a>				X	X	X	Compare dwords ES:[(E)DI] (second operand) with [(E)SI] (first operand)
A6	CMPSB	X	X	X	X	X	X	Compare bytes ES:[(E)DI] with DS:[(E)SI]
A7	CMPSW	X	X	X	X	X	X	Compare words ES:[(E)DI] with DS:[(E)SI]
A7	CMPSD				X	X	X	Compare dwords ES:[(E)DI] with DS:[(E)SI]

---

## Description

The CMPS instruction compares the byte, word, or doubleword pointed to by the source-index register with the byte, word, or doubleword pointed to by the destination-index register.

If the address-size attribute of this instruction is 16-bits, the SI and DI registers will be used for source- and destination-index registers; otherwise the ESI and EDI registers will be used. Load the correct index values into the SI and DI (or ESI and EDI) registers before running the CMPS instruction.

The comparison is done by subtracting the operand indexed by the destination-index register from the operand indexed by the source-index register.

Note that the direction of subtraction for the CMPS instruction is [SI] - [DI] or [ESI] - [EDI]. The left operand (SI or ESI) is the source and the

right operand (DI or EDI) is the destination. This is the reverse of the usual Intel convention in which the left operand is the destination and the right operand is the source.

The result of the subtraction is not stored; only the flags reflect the change. The types of the operands determine whether bytes, words, or doublewords are compared. For the first operand (SI or ESI), the DS register is used, unless a segment override byte is present. The second operand (DI or EDI) must be addressable from the ES register; no segment override is possible.

After the comparison is made, both the source-index register and destination-index register are automatically advanced. If the DF flag is 0 (a CLD instruction was run), the registers increment; if the DF flag is 1 (an STD instruction was run), the registers decrement. The registers increment or decrement by 1 if a byte is compared, by 2 if a word is compared, or by 4 if a doubleword is compared.

The CMPSB, CMPSW and CMPSD instructions are synonyms for the byte, word, and doubleword CMPS instructions, respectively.

The CMPS instruction can be preceded by the REPE or REPNE prefix for block comparison of CX or ECX bytes, words, or doublewords. Refer to the description of the REP instruction for more information on this operation.

---

## Operation

```
IF (instruction = CMPSD) OR
  (instruction has operands of type DWORD)
  THEN OperandSize ` 32;
ELSE OperandSize ` 16;
FI;
IF AddressSize = 16
  THEN
    use SI for source-index and DI for destination-index
  ELSE (* AddressSize = 32 *)
    use ESI for source-index and EDI for destination-index;
  FI;
IF byte type of instruction
  THEN
    set ZF based on
      [source-index] - [destination-index]; (* byte comparison *)
    IF DF = 0 THEN IncDec ` 1 ELSE IncDec ` -1; FI;
  ELSE
    IF OperandSize = 16
      THEN
        set ZF based on
          [source-index] - [destination-index]; (* word comparison *)
        IF DF = 0 THEN IncDec ` 2 ELSE IncDec ` -2; FI;
      ELSE (* OperandSize = 32 *)
        set ZF based on
          [source-index] - [destination-index]; (* dword comparison *)
        IF DF = 0 THEN IncDec ` 4 ELSE IncDec ` -4; FI;
      FI;
    FI;
  FI;
source-index = source-index + IncDec;
destination-index = destination-index + IncDec;
```

---

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
*			*	*	*	*	*

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

---

# Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

# Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

# Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

# Related Information

- [Description](#)
- [Flags Affected](#)
- [Operation](#)
- [Protected Mode Exceptions](#)
- [Real Address Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

# CMPXCHG-Compare and Exchange

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F B0 /r	CMPXCHG r/m8,r8	X	X					Compare AL with r/m byte. If equal, set ZF and load byte reg into r/m byte. Else, clear ZF and load r/m byte into AL
0F B1 /r	CMPXCHG r/m16,r16	X	X					Compare AX with r/m word. If equal, set ZF and load word reg into r/m word. Else, clear ZF and load r/m word into AX

0F B1 /r CMPXCHG r/m32,r32

X X Compare EAX with r/m dword. If equal, set ZF and load dword reg into r/m dword. Else, clear ZF and load r/m dword into EAX.

---

## Description

The CMPXCHG instruction compares the accumulator (AL, AX, or EAX register) with DEST. If they are equal, SRC is loaded into DEST. Otherwise, DEST is loaded into the accumulator.

---

## Operation

```
IF accumulator=DEST
    ZF ` 1
    DEST ` SRC
ELSE
    ZF ` 0
    accumulator ` DEST
```

---

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
*			*	*	*	*	*

The CF, PF, AF, SF, and OF flags are affected as if a CMP instruction had been run with DEST and the accumulator as operands. The ZF flag is set if the destination operand and the accumulator are equal; otherwise it is cleared.

---

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH.

---

# Virtual 8086 Mode Exceptions

Same exceptions as in real-address mode; #PF (fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Notes

This instruction can be used with a LOCK prefix. In order to simplify interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. DEST is written back if the comparison fails, and SRC is written into the destination otherwise. (The processor never produces a locked read without also producing a locked write.) This instruction is not supported on Intel386 processors.

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

## CMPXCHG8B-Compare and Exchange 8 Bytes

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F C7 /1	CMPXCHG8B <a href="#">r/m64</a>							<a href="#">X</a> Compare EDX:EAX with r/m qword. If equal, set ZF and load ECX:EBX into r/m qword. Else, clear ZF and load r/m qword into EDX:EAX

## Description

The CMPXCHG8B instruction compares the 64-bit value in EDX:EAX with DEST. EDX contains the high-order 32 bits, and EAX contains the low-order 32 bits of 64-bit value. If they are equal, the 64-bit value in ECX:EBX is stored into DEST. ECX contains the high-order 32 bits and EBX contains the low-order 32 bits. Otherwise, DEST is loaded into EDX:EAX.

---

## Operation

```
IF EDX:EAX=DEST
    ZF ` 1
    DEST ` ECX:EBX
ELSE
    ZF ` 0
    EDX:EAX ` DEST
```

---

## Flags Affected

OF DF IF SF ZF AF PF CF

\*

The ZF flag is set if the destination operand and EDX:EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are unaffected.

---

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

The destination operand must be a memory operand, not a register. If the CMPXCHG8B instruction is run with a modr/m byte representing a register as the destination operand, #UD occurs.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in real-address mode; #PF (fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3. #UD if modr/m byte represents a register as the destination.

---

# Notes

This instruction can be used with a LOCK prefix. In order to simplify interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. DEST is written back if the comparison fails, and SRC is written into the destination otherwise. (The processor never produces a locked read without also producing a locked write.)

The "r/m64" syntax had previously been used only in the context of floating point operations. It indicates a 64-bit value, in memory at an address determined by the modr/m byte. This instruction is not supported on Intel486 processors.

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

## CPUID-CPU Identification

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F A2	CPUID						X	EAX ` CPU identification information

## Description

The CPUID instruction provides information to software about the vendor, family, model, and stepping of microprocessor on which it is running. An input value loaded into the EAX register for this instruction indicates what information should be returned by the CPUID instruction.

Following processing of the CPUID instruction with a zero in EAX, the EAX register contains the highest input value understood by the CPUID instruction. For the Pentium processor, the value in EAX will be a one. Also included in the output of this instruction with an input value of zero in EAX is a vendor identification string contained in the EBX, EDX, and ECX registers. EBX contains the first four characters, EDX contains

the next four characters and ECX contains the last four characters. For Intel processors, the vendor identification string is "GenuineIntel" as follows:

```
EBX ` 756e6547h (* "Genu", with G in the low nibble of BL *)
EDX ` 49656e69h (* "ineI", with i in the low nibble of DL *)
ECX ` 6c65746eh (* "ntel", with n in the low nibble of CL *)
```

Following processing of the CPUID instruction with an input value of one loaded into the EAX register, EAX[3:0] contains the stepping id of the microprocessor, EAX[7:4] contains the model (the first model will be indicated by 0001B in these bits) and EAX[11:8] contains the family (5 for the Pentium processor family). EAX[31:12] are reserved, as well as EBX, and ECX. The Pentium processor sets the feature register, EDX, to 1BFH indicating which features the Pentium processor supports. A feature flag set to one indicates that the corresponding feature is supported. The feature set register is defined as follows:

```
EDX[0:0] FPU on chip
EDX[6:1] Refer to the Intel documentation
EDX[7:7] Machine Check Exception
EDX[8:8] CMPXCHG8B Instruction
EDX[31:9] Reserved
```

Software should determine the vendor identification in order to properly interpret the feature register flag bits. For more information on the feature set register, see the Intel documentation.

---

## Operation

```
switch (EAX)
case 0:
    EAX ` hv;(* hv=1 for the Pentium processor *)
    (* hv is the highest input value that is understood by CPUID. *)

    EBX ` Vendor identification string;
    EDX ` Vendor identification string;
    ECX ` Vendor identification string;
    break;

case 1:
    EAX[3:0] ` Stepping ID;
    EAX[7:4] ` Model;
    EAX[11:8] ` Family;
    EAX[31:12] ` Reserved;

    EBX ` reserved;    (* 0 *)
    ECX ` reserved;    (* 0 *)
    EBX ` feature flags;
    break;

default: (* EAX > hv *)
    EAX ` reserved, undefined;
    EBX ` reserved, undefined;
    ECX ` reserved, undefined;
    EDX ` reserved, undefined;
    break;
end-of-switch
```

---

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

# CWD/CDQ-Convert Word to Double/Convert Double to Quad

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
99	CWD	X	X	X	X	X	X	DX ` sign-extend of AX
99	CDQ			X	X	X		EDX ` sign-extend of EAX

## Description

CWD and CDQ double the size of the source operand. The CWD instruction copies the sign (bit 15) of the word in the AX register into every bit position in the DX register. The CDQ instruction copies the sing (bit 31) of the doubleword in the EAX register into every bit position in the EDX register. The CWD instruction can be used to produce a doubleword dividend from a word before a word division, and the CDQ instruction can be used to produce a quadword dividend from a doubleword before doubleword division. The CWD and CDQ instructions are different mnemonics for the same opcode. Which one gets run is determined by whether it is in a 16- or 32-bit segment and the presence of any operand-size override prefixes.

## Operation

```
IF OperandSize = 16 (* instruction = CWD *)
THEN DX ` SignExtend(AX);
ELSE (* OperandSize = 32, instruction = CDQ *)
  EDX ` SignExtend(EAX);
FI;
```

## Related Information

- [Description](#)
- [Flags Affected](#)
- [Operation](#)
- [Protected Mode Exceptions](#)

---

# CWDE-Convert Word to Doubleword

See entry for CBW/CWDE-Convert Byte to Word/Convert Word to Doubleword.

---

# DAA-Decimal Adjust AL after Addition

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
27	DAA	X	X	X	X	X	X	Decimal adjust AL after addition

---

## Description

Run the DAA instruction only after running an ADD instruction that leaves a two-BCD-digit byte result in the AL register. The ADD operands should consist of two packaged BCD digits. The DAA instruction adjusts the AL register to contain the correct two-digit packed decimal result.

---

## Operation

```
IF (((AL AND 0FH) > 09H) or EFLAGS.AF = 1)
THEN
    AL ` AL + 06H;
FI;
IF (((AL AND 0F0H) > 90H) or EFLAGS.CF = 1)
THEN
    AL ` AL + 60H;
    CF ` 1;
FI;
```

---

## Flags Affected

OF DF IF SF ZF AF PF CF  
? \* \* \* \* \*

The AF and CF flags are set if there is a decimal carry, cleared if there is no decimal carry; the SF, ZF and PF flags are set according to the result. The OF flag is undefined.

## Related Information

- Description
- Flags Affected
- Operation
- Protected Mode Exceptions
- Protected Mode Exceptions
- Virtual 8086 Mode Exceptions

## DAS-Decimal Adjust AL after Subtraction

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
2F	DAS	X	X	X	X	X	X	Decimal adjust AL after subtraction

## Description

Run the DAS instruction only after a subtraction instruction that leaves a two-BCD-digit byte result in the AL register. The operands should consist of two packed BCD digits. The DAS instruction adjusts the AL register to contain the correct packed two-digit decimal result.

## Operation

```
tmpCF ` 0;
tmpAL ` AL;
IF (((tmpAL AND 0FH) > 9H) or AF = 1)
THEN
  AF ` 1;
  AL ` AL - 6H;
  tmpCF ` (AL < 0) OR CF;
FI;
IF ((tmpAL > 99H) or CF = 1)
THEN
  AL ` AL - 60H;
  tmpCF ` 1;
FI;
CF ` tmpCF;
```

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
?			*	*	*	*	*

The AF and CF flags are set if there is a decimal borrow, cleared if there is no decimal borrow; the SF, ZF and PF flags are set according to the result. The OF flag is undefined.

## Related Information

- Description
- Flags Affected
- Operation
- Protected Mode Exceptions
- Protected Mode Exceptions
- Virtual 8086 Mode Exceptions

## DEC-Decrement by 1

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
FE /1	DEC <a href="#">r/m8</a>	X	X	X	X	X	X	Decrement <a href="#">r/m</a> byte by 1

FF /1	DEC <code>r/m16</code>	<code>X X X X X X</code> Decrement r/m word by 1
FF /1	DEC <code>r/m32</code>	<code>X X X</code> Decrement r/m dword by 1
48+rw	DEC <code>r16</code>	<code>X X X X X X</code> Decrement word register by 1
48+rd	DEC <code>r32</code>	<code>X X X</code> Decrement dword register by 1

-----

## Description

The DEC instruction subtracts 1 from the operand. The DEC instruction does not change the CF flag. To affect the CF flag, use the SUB instruction with an immediate operand of 1.

-----

## Operation

DEST`DEST - 1;

-----

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
*			*	*	*	*	

The OF, SF, ZF, AF, and PF flags are set according to the result.

-----

## Protected Mode Exceptions

#GP(0) if the result is a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

-----

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Related Information

- [Description](#)
- [Flags Affected](#)
- [Operation](#)
- [Protected Mode Exceptions](#)
- [Real Address Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

## DIV-Unsigned Divide

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
F6 /6	DIV <a href="#">r/m8</a>	X	X	X	X	X	X	Unsigned divide AX by r/m byte (AL=Quotient, AH=Remainder)
F7 /6	DIV <a href="#">r/m16</a>	X	X	X	X	X	X	Unsigned divide DX:AX by r/m word (AX=Quotient, DX=Remainder)
F7 /6	DIV <a href="#">r/m32</a>	X	X	X				Unsigned divide EDX:EAX by r/m dword (EAX=Quotient, EDX=Remainder)

## Description

The DIV instruction performs an unsigned division. The dividend is implicit; only the divisor is given as an operand. The remainder is always less than the divisor. The type of the divisor determines which registers to use as follows:

SIZE	DIVIDEND	DIVISOR	QUOTIENT	REMAINDER
byte	AX	<i>r/m8</i>	AL	AH
word	DX:AX	<i>r/m16</i>	AX	DX
dword	EDX:EAX	<i>r/m32</i>	EAX	EDX

---

## Operation

```
temp ← dividend / divisor;
IF temp does not fit in quotient
THEN Interrupt 0;
ELSE
  quotient ← temp;
  remainder ← dividend MOD (r/m);
FI;
```

**Note:** Divisions are unsigned. The divisor is given by the *r/m* operand. The dividend, quotient, and remainder use implicit registers. Refer to the table under "Description".

---

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
?			?	?	?	?	?

The OF, SF, ZF, AF, PF, and CF flags are undefined.

---

## Protected Mode Exceptions

Interrupt 0 if the quotient is too large to fit in the designated register (AL, AX, or EAX), or if the divisor is 0; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 0 if the quotient is too big to fit in the designated register (AL, AX, or EAX), or if the divisor is 0; Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

# ENTER-Make Stack Frame for Procedure Parameters

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
C8 iw 00	ENTER <a href="#">imm16</a> ,0	X	X	X	X	X	X	Make procedure stack frame
C8 iw 01	ENTER <a href="#">imm16</a> ,1	X	X	X	X	X	X	Make stack frame for procedure parameters
C8 iw ib	ENTER <a href="#">imm16</a> , <a href="#">imm8</a>	X	X	X	X	X	X	Make stack frame for procedure parameters

---

## Description

The ENTER instruction creates the stack frame required by most block-structured high-level languages. The first operand specifies the number of bytes of dynamic storage allocated on the stack for the routine being entered. The second operand gives the lexical nesting level (0 to 31) of the routine within the high-level language source code. It determines the number of stack frame pointers copied into the new stack frame from the preceding frame.

Both the operand-size attribute and the stack-size attribute are used to determine whether BP or EBP is used for the current frame pointer and SP or ESP is used for the stack pointer.

If the operand-size attribute is 16-bits, the processor uses the BP register as the frame pointer and the SP register as the stack pointer, unless the stack-size attribute is 32-bits in which case it uses EBP for the frame pointer and ESP for the stack pointer. If the operand-size attribute is 32-bits, the processor uses the EBP register for the frame pointer and the ESP register for the stack pointer, unless the stack-size attribute is 16-bits in which case it uses BP for the frame pointer and SP for the stack pointer.

If the second operand is 0, the ENTER instruction pushes the frame pointer (BP or EBP register) onto the stack; the ENTER instruction then subtracts the first operand from the stack pointer and sets the frame pointer to the current stack-pointer value.

For example, a procedure with 12 bytes of local variables would have an ENTER 12,0 instruction at its entry point and a LEAVE instruction before every RET instruction. The 12 local bytes would be addressed as negative offsets from the frame pointer.

---

## Operation

```

level ` level MOD 32
2ndOperand <- 2ndOperand MOD 32
IF operand_size = 16 THEN Push(bp) ELSE Push(ebp) FI;
IF stkSize = 16 THEN framePtr = sp ELSE framePtr = esp FI;
FOR i ` 1 TO (2ndOperand - 1)
DO
  IF oprand_size = 16
  THEN
    IF stkSize = 16
    THEN
      bp = bp - 2
      Push([bp]) (* word push *)
    ELSE (* stkSize = 32 *)
      ebp = ebp - 2
      Push([ebp]) (* word push *)
    FI;
  ELSE (* operand_size = 32 *)
    IF stkSize = 16
    THEN
      bp = bp - 4
      Push([bp]) (* doubleword push *)
    ELSE (* stkSize = 32 *)
      ebp = ebp - 4
      Push([ebp]) (* doubleword push *)
    FI;
  FI;
OD;
IF stkSize = 16
THEN Push(framePtr); (* word push *)
ELSE Pushd(framePtr); (* doubleword push *)
FI;
IF stkSize = 16
THEN
  bp = framePtr
  sp = sp - 1stOperand
ELSE
  ebp = framePtr
  esp = esp - 1stOperand
FI;

```

-----

## Protected Mode Exceptions

#SS(0) if the SP or ESP value would exceed the stack limit at any point during instruction processing; #PF(fault-code) for a page fault.

-----

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Protected Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

-----

## F2XM1-Compute (2 to the power of X) minus 1

---

=====

-----

=====

-----

-----

=====

# Notes

If the operand is outside the acceptable range, the result of F2XM1 is undefined.

Values other than 2 can be exponentiated using the formula

$xy = 2^{(y * \log_2 x)}$

The instructions FLDL2T and FLDL2E load the constants  $\log_2 10$  and  $\log_2 e$ , respectively. FYL2X can be used to calculate  $y * \log_2 x$  for arbitrary positive x.

# Related Information

- Description
- FPU Flags Affected
- Notes
- Operation
- Numeric Exceptions
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

# FABS-Absolute Value

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 E1	FABS	X	X	X	X	X	X	Replace ST with its absolute value

# Description

The absolute value instruction clears the sign bit of ST. This operation leaves a positive value unchanged, or replaces a negative value with a positive value of equal magnitude.

---

## Operation

sign bit of ST ` 0

---

## Numeric Exceptions

IS

---

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Notes

The invalid-operation exception is raised only on stack underflow. No exception is raised if the operand is a signalling NaN or is in an unsupported format.

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

---

# FADD/FADDP/FIADD-Add

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
DE C1	FADD	X	X	X	X	X	X	Add ST to ST(1) and pop ST
D8 C0+i	FADD ST,ST(i)	X	X	X	X	X	X	Add ST(i) to ST
DC C0+i	FADD ST(i),ST	X	X	X	X	X	X	Add ST to ST(i)
D8 /0	FADD m32real	X	X	X	X	X	X	Add m32real to ST
DC /0	FADD m64real	X	X	X	X	X	X	Add m64real to ST
DE C0+i	FADDP ST(i),ST	X	X	X	X	X	X	Add ST to ST(i) and pop ST
DE /0	FIADD m16int	X	X	X	X	X	X	Add m16int to ST
DA /0	FIADD m32int	X	X	X	X	X	X	Add m32int to ST

---

## Description

The addition instructions add the source and destination operands and return the sum to the destination. The operand at the stack top can be doubled by coding:

FADD ST, ST(0)

---

## Operation

DEST ← DEST + SRC;  
If instruction = FADDP THEN pop ST FI;

---

## Numeric Exceptions

P, U, O, D, I, IS.

---

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH; Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF (fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

If the source operand is in memory, it is automatically converted to the extended-real format.

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## FBLD-Load Binary Coded Decimal

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
DF /4	FBLD <code>m80bcd</code>	X	X	X	X	X	X	Push <code>m80bcd</code> onto the FPU stack

## Description

FBLD converts the BCD source operand into extended-real format, and pushes it onto the FPU stack. See Figure 6-10 for BCD data layout.

## Operation

Decrement FPU stack-top pointer;  
ST(0) ← SRC;

## Numeric Exceptions

IS.

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH; Interrupt 7 if either EM or TS in CR0 is set.

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF (fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

# Notes

The source is loaded without rounding error. The sign of the source is preserved, including the case where the value is negative zero.

The packed decimal digits are assumed to be in the range 0-9. The instruction does not check for invalid digits (A-FH), and the result of attempting to load an invalid encoding is undefined.

ST(7) must be empty to avoid causing an invalid-operation exception.

## Related Information

- Description
- FPU Flags Affected
- Notes
- Operation
- Numeric Exceptions
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

## FBSTP-Store Binary Coded Decimal and Pop

### Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
DF /6	FBSTP <code>m80bcd</code>	X	X	X	X	X	X	Store ST in m80bcd and pop ST

### Description

FBSTP converts the value in ST into a packed decimal integer, stores the result at the destination in memory, and pops ST. Non-integral values are first rounded according to the RC field of the control word. See Figure 6-10 for BCD data layout.

### Operation

DEST ` ST(0);  
pop ST;

-----

## Numeric Exceptions

P, I, IS.

-----

## Protected Mode Exceptions

#GP(0) if the destination is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH; Interrupt 7 if either EM or TS in CR0 is set.

-----

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF (fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

-----

## FCHS-Change Sign

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 E0	FCHS	X	X	X	X	X	X	Replace ST with a value of opposite sign

# Description

The change sign instruction inverts the sign bit of ST. This operation replaces a positive value with a negative value of equal magnitude, or vice-versa.

# Operation

sign bit of ST  $\neg$  NOT (sign bit of ST)

# Numeric Exceptions

IS

# Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

# Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CRO is set.

# Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

# Notes

The invalid-operation exception is raised only on stack underflow, even if the operand is signalling NaN or is an unsupported format.

## Related Information

- Description
- FPU Flags Affected
- Notes
- Operation
- Numeric Exceptions
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

## FCLEX/FNCLEX-Clear Exceptions

### Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
9B DB E2	FCLEX	X	X	X	X	X	X	Clear FP exception flags after checking for FP error conditions
DB E2	FNCLEX	X	X	X	X	X	X	Clear FP exeption flags without checking for FP error conditions

### Description

FCLEX clears the exception flags, the exception status flag, and the busy flag of the FPU status word.

### Operation

SW[0..7] `0;  
SW[15] `0;

-----

## Numeric Exceptions

None

-----

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

-----

## Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

-----

## Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

-----

## Notes

FCLEX checks for unmasked floating-point error conditions before clearing the exception flags, FNCLEX does not.

-----

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

# FCOM/FCOMP/FCOMPP-Compare Real

---

## Details Table

Encoding	Instruction	0 1 2 3 4 5	Description
D8 D1	FCOM	X X X X X X	Compare ST with ST(1)
D8 D0+i	FCOM ST(i)	X X X X X X	Compare ST with ST(i)
D8 /2	FCOM m32real	X X X X X X	Compare ST with m32real
DC /2	FCOM m64real	X X X X X X	Compare ST with m64real
D8 D9	FCOMP	X X X X X X	Compare ST with ST(1) and pop ST
D8 D8+i	FCOMP ST(i)	X X X X X X	Compare ST with ST(i) and pop ST
D8 /3	FCOMP m32real	X X X X X X	Compare ST with m32real and pop ST
DC /3	FCOMP m64real	X X X X X X	Compare ST with m64real and pop ST
DE D9	FCOMPP	X X X X X X	Compare ST with ST(1) and pop ST twice

---

## Description

The compare real instructions compare the stack top to the source, which can be a register or a single- or double-real memory operand. If no operand is encoded, ST is compared to ST(1). Following the instruction, the condition codes reflect the relation between ST and the source operand.

---

## Operation

Case (relation of operands) OF

Not comparable: C3, C2, C0` 111;

ST > SRC: C3, C2, C0` 000;

ST < SRC: C3, C2, C0` 001;

ST = SRC: C3, C2, C0` 100;

If instruction = FCOMP THEN pop ST; FI;

If instruction = FCOMPP THEN pop ST; pop ST; FI;

---

## FPU Flags Affected

FPU FLAGS	EFLAGS
C0	CF
C1	None
C2	PF
C3	ZF

C1 as described in [FPU Flags Affected](#); C0, C2, C3 as specified above.

---

## Numeric Exceptions

D, I, IS.

---

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF (fault-code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside effective address space from 0 to 0FFFFH; Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF (fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

If either operand is NaN or is in an undefined format, or if a stack fault occurs, the invalid-operation exception is raised, and the condition bits are set to "unordered."

The sign of zero is ignored, so that -0.0 == +0.0.

---

## Related Information

- Description
- FPU Flags Affected
- Notes
- Operation
- Numeric Exceptions
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

# FCOS-Cosine

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 FF	FCOS		X	X	X			Replace ST with its cosine

## Description

The cosine instruction replaces the contents of ST with cos(ST). ST, expressed in radians, must lie in the range  $101 < 2^{63}$ .

## Operation

IF operand is in range  
THEN  
    C2 ` 0;  
    ST ` cos(ST);  
ELSE  
    C2 ` 1;  
FI;

## FPU Flags Affected

C0 C1 C2 C3

? \* ? \*

C1, C2 as described in [FPU Flags Affected](#); C0, C3 undefined.

---

## Numeric Exceptions

P,D,I,IS.

---

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Notes

If the operand is outside the acceptable range, the C2 flag is set, and ST remains unchanged. It is the programmer's responsibility to reduce the operand to an absolute value smaller than 2<sup>63</sup> by subtracting an appropriate integer multiple of 2<sup>0</sup>. See the Intel documentation for discussion of the proper value to use for  $\bar{O}$  in performing such reductions.

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

# FDECSTP-Decrement Stack-Top Pointer

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 F6	FDECSTP	X	X	X	X	X	X	Decrement top-of-stack pointer for FPU register stack

## Description

FDESCSTP subtracts one (without carry) from the three-bit TOP field of the FPU status word.

## Operation

IF TOP=0  
THEN TOP ` 7;  
ELSE TOP ` TOP-1;  
FI;

## Numeric Exceptions

None.

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

# Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

## Notes

The effect of FDECSTP is to rotate the stack. It does not alter register tags or contents, nor does it transfer data.

## Related Information

- Description
- FPU Flags Affected
- Notes
- Operation
- Numeric Exceptions
- Protected Mode Exceptions
- Real Address Mode Exceptions

## FDISI/FNDISI-Disable Interrupts

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
9B DB E1	FDISI	X	X					Sets the interrupt enable mask in the control word
DB E1	FNDISI	X	X					Sets the interrupt enable mask in the control word

## Description

Operation

FPU Flags Affected

C0 C1 C2 C3

Numeric Exceptions

Protected Mode Exceptions

Real Address Mode Exceptions

Notes

Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

---

# FDIV/FDIVP/FIDIV-Divide

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D8 /6	FDIV <a href="#">m32real</a>	X	X	X	X	X	X	Replace ST with $ST \div m32real$
DC /6	FDIV <a href="#">m64real</a>	X	X	X	X	X	X	Replace ST with $ST \div m64real$
D8 F0+i	FDIV ST,ST(i)	X	X	X	X	X	X	Replace ST with $ST \div ST(i)$
DC F8+i	FDIV <a href="#">ST(i)</a> ,ST	X	X	X	X	X	X	Replace ST(i) with $ST(i) \div ST$
DE F8+i	FDIVP <a href="#">ST(i)</a> ,ST	X	X	X	X	X	X	Replace ST(i) with $ST(i) \div ST$ ; pop ST
DE F9	FDIV	X	X	X	X	X	X	Replace ST(1) with $ST(1) \div ST$ ; pop ST
DE /6	FIDIV <a href="#">m16int</a>	X	X	X	X	X	X	Replace ST with $ST \div m16int$
DA /6	FIDIV <a href="#">m32int</a>	X	X	X	X	X	X	Replace ST with $ST \div m32int$

---

## Description

The division instructions divide the stack top by other operand and return the quotient to the destination.

---

## Operation

FDIV DEST, SRC  
DEST`DEST ÷ SRC  
If instruction = FDIVP THEN pop ST FI;

---

# Numeric Exceptions

P, U, O, Z, D, I, IS.

---

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH; Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

If the source operand is in memory, it is automatically converted to the extended-real format.

The performance of the division instructions depends on the PC (Precision Control) field of the FPU control word. If PC specifies a precision of 53 bits, the division instructions will run in 33 clocks. If the specified precision is 24 bits, the division instructions will take only 19 clocks.

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

# FDIVR/FDIVRP/FIDIVR-Reverse Divide

---

## Details Table

Encoding	Instruction	0 1 2 3 4 5	Description
D8 /7	FDIVR <i>m32real</i>	X X X X X X	Replace ST with <i>m32real</i> ÷ ST
DC /7	FDIVR <i>m64real</i>	X X X X X X	Replace ST with <i>m64real</i> ÷ ST
D8 F8+i	FDIVR ST,ST( <i>i</i> )	X X X X X X	Replace ST with ST( <i>i</i> ) ÷ ST
DC F0+i	FDIVR ST( <i>i</i> ),ST	X X X X X X	Replace ST( <i>i</i> ) with ST ÷ ST( <i>i</i> )
DE F0+i	FDIVRP ST( <i>i</i> ),ST	X X X X X X	Replace ST( <i>i</i> ) with ST ÷ ST( <i>i</i> ); pop ST
DE F1	FDIVR	X X X X X X	Replace ST(1) with ST ÷ ST(1); pop ST
DE /7	FIDIVR <i>m16int</i>	X X X X X X	Replace ST with <i>m16int</i> ÷ ST
DA /7	FIDIVR <i>m32int</i>	X X X X X X	Replace ST with <i>m32int</i> ÷ ST

---

## Description

The division instructions divide the other operand by the stack top and return the quotient to the destination.

---

## Operation

FDIVR DEST, SRC  
DEST ← SRC ÷ DEST  
IF instruction = FDIVRP THEN pop ST FI;

---

## Numeric Exceptions

P, U, O, Z, D, I, IS.

---

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH; Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

If the source operand is in memory, it is automatically converted to the extended-real format.

The performance of the division instructions depends on the PC (Precision Control) field of the FPU control word. If PC specifies a precision of 53 bits, the reverse division instructions will run in 33 clocks. If the specified precision is 24 bits, the reverse division instructions will take only 19 clocks.

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## FENI/FNENI-Enable Interrupts

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
9B DB E0	FENI	X	X					Clears the interrupt control mask in the control word
DB E0	FNENI	X	X					Clears the interrupt enable mask in the control word

-----

Description

-----

Operation

-----

FPU Flags Affected

C0 C1 C2 C3

-----

Numeric Exceptions

-----

Protected Mode Exceptions

-----

Real Address Mode Exceptions

---

## Notes

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

---

## FFREE-Free Floating-Point Register

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
DD C0+i	FFREE <a href="#">ST(i)</a>	<a href="#">X</a>	<a href="#">X</a>	<a href="#">X</a>	<a href="#">X</a>	<a href="#">X</a>	<a href="#">X</a>	Tag ST(i) as <code>empty</code>

---

## Description

FFREE tags the destination register as *empty*.

---

## Operation

TAG(i) ` 11B;

-----

## Numeric Exceptions

None

-----

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

-----

## Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

-----

## Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

-----

## Notes

FFREE does not affect the contents of the destination register. The floating-point stack pointer (TOP) is also unaffected.

-----

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

-----

# FICOM/FICOMP-Compare Integer

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
DE /2	FICOM <code>m16int</code>	X	X	X	X	X	X	Compare ST with m16int
DA /2	FICOM <code>m32int</code>	X	X	X	X	X	X	Compare ST with m32int
DE /3	FICOMP <code>m16int</code>	X	X	X	X	X	X	Compare ST with m16int and pop ST
DA /3	FICOMP <code>m32int</code>	X	X	X	X	X	X	Compare ST with m32int and pop ST

## Description

The compare integer instructions compare the stack top to the source. Following the instruction, the condition codes reflect the relation between ST and the source operand.

## Operation

CASE (relation of operands) OF  
Not comparable: C3, C2, C0`111;  
ST > SRC: C3, C2, C0`000;  
ST < SRC: C3, C2, C0`001;  
ST = SRC: C3, C2, C0`100;  
If instruction = FICOMP THEN pop ST; FI;

## FPU Flags Affected

FPU FLAGS	EFLAGS
C0	CF
C1	(none)
C2	PF
C3	ZF

C1 as described in [FPU Flags Affected](#); C0, C2, C3 as specified above.

---

## Numeric Exceptions

D, I, IS.

---

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH; Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

The memory operand is converted to extended-real format before the comparison performed.

If either operand is a NaN or is in an undefined format, or if a stack fault occurs, the invalid operation exception is raised, and the condition bits are set to "unordered."

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

# FILD-Load Integer

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
DF /0	FILD <code>m16int</code>	X	X	X	X	X	X	Push <code>m16int</code> onto the FPU stack
DB /0	FILD <code>m32int</code>	X	X	X	X	X	X	Push <code>m32int</code> onto the FPU stack
DF /5	FILD <code>m64int</code>	X	X	X	X	X	X	Push <code>m64int</code> onto the FPU stack

---

## Description

FILD converts the source signed integer operand into extended-real format, and pushes it onto the FPU stack.

---

## Operation

Decrement FPU stack-top pointer;  
ST(0) ← SRC;

---

## Numeric Exceptions

IS.

---

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH; Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

The source is loaded without rounding error.

ST(7) must be empty to avoid causing an invalid-operation exception.

---

## Related Information

- [Description](#)
- [FPU Flags Affected](#)
- [Notes](#)
- [Operation](#)
- [Numeric Exceptions](#)
- [Protected Mode Exceptions](#)
- [Real Address Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

---

## FINCSTP-Increment Stack-Top Pointer

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 F7	FINCSTP	X	X	X	X	X	X	Increment top-of-stack pointer for FPU register stack

---

## Description

FINCSTP adds one (without carry) to the three-bit TOP field of the FPU status word.

---

## Operation

```
IF TOP = 7  
THEN TOP ← 0;  
ELSE TOP ← TOP + 1;  
FI;
```

---

## Numeric Exceptions

None

---

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

#NM is either EM or TS in CR0 is set.

---

## Notes

The effect of FINCSTP is to rotate the stack. It does not alter register tags or contents, nor does it transfer data. It is not equivalent to popping the stack, because it does not set the tag of the old stack-top to *empty*.

---

## Related Information

- Description
- FPU Flags Affected
- Notes
- Operation
- Numeric Exceptions
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

# FINIT/FNINIT-Initialize Floating-Point Unit

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
9B DB E3	FINIT	X	X	X	X	X	X	Initialize FPU after checking for unmasked FP error condition
DB E3	FNINIT	X	X	X	X	X	X	Initialize FPU without checking for unmasked FP error condition

## Description

The initialization instructions set the FPU into a known state, unaffected by any previous activity.

The FPU control word is set to 037FH (round to nearest, all exceptions masked, 64-bit precision). The status word is cleared (no exception flags set, stack register R0=stack-top). The stack registers are all tagged as *empty*. The error pointers (both instruction and data) are cleared.

## Operation

CW ` 037FH; (\* Control word \*)  
SW ` 0; (\*Status word\*)  
TW ` FFFFH; (\* Tag word \*)  
FEA ` 0; FDS ` 0; (\* Data pointer \*)  
FIP ` 0; FOP ` 0; FCS ` 0; (\* Instruction pointer \*)

# FPU Flags Affected

C0	C1	C2	C3
0	0	0	0

C0, C1, C2, C3 cleared.

-----

## Numeric Exceptions

None

-----

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

-----

## Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

-----

## Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

-----

## Notes

FINIT checks for unmasked floating-point error conditions before performing the initialization; FNINIT does not.

On the Pentium processor, unlike the Intel387 math coprocessor, FINIT and FNINIT clear the error pointers.

-----

## Related Information

[Description](#)

[FPU Flags Affected](#)

# FIST/FISTP-Store Integer

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
DF /2	FIST <a href="#">m16int</a>	X	X	X	X	X	X	Store ST in m16int
DB /2	FIST <a href="#">m32int</a>	X	X	X	X	X	X	Store ST in m32int
DF /3	FISTP <a href="#">m16int</a>	X	X	X	X	X	X	Store ST in m16int and pop ST
DB /3	FISTP <a href="#">m32int</a>	X	X	X	X	X	X	Store ST in m32int and pop ST
DF /7	FISTP <a href="#">m64int</a>	X	X	X	X	X	X	Store ST in <a href="#">m64int</a> and pop ST

## Description

FIST converts the value in ST into a signed integer according to the RC field of the control word and transfers the result to the destination. ST remains unchanged. FIST accepts word and short integer destinations; FISTP accepts these and long integers as well.

## Operation

DEST ← ST(0);  
IF instruction = FISTP THEN pop ST FI;

## Numeric Exceptions

P, I, IS.

---

## Protected Mode Exceptions

#GP(0) if the destination is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH; Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

Negative zero is stored with the same encoding (00..00) as positive zero. If the value is too large to represent as an integer, an I exception is raised. The masked response is to write the most negative integer to memory.

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## FLD-Load Real

---

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 /0	FLD <a href="#">m32real</a>	X	X	X	X	X	X	Push m32real onto the FPU stack
DD /0	FLD <a href="#">m64real</a>	X	X	X	X	X	X	Push m64real onto the FPU stack
DB /5	FLD <a href="#">m80real</a>	X	X	X	X	X	X	Push m80real onto the FPU stack
D9 C0+i	FLD <a href="#">ST(i)</a>	X	X	X	X	X	X	Push ST(i) onto the FPU stack

## Description

FLD pushes the source operand onto the FPU stack. If the source is a register, the register number used is that before the stack-top pointer is decremented. In particular, coding FLD ST(0)

duplicates the stack top.

## Operation

Decrement FPU stack-top pointer;  
ST(0) ← SRC;

## Numeric Exceptions

D, I, IS.

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH; Interrupt 7 if either EM or TS in CR0 is set.

-----

# Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Notes

If the source operand is single- or double-real format, it is automatically converted to the extended-real format. Loading an extended-real operand does not require conversion, so the I and D exceptions will not occur in this case.

ST(7) must be empty to avoid causing an invalid-operation exception.

-----

## Related Information

- [Description](#)
  - [FPU Flags Affected](#)
  - [Notes](#)
  - [Operation](#)
  - [Numeric Exceptions](#)
  - [Protected Mode Exceptions](#)
  - [Real Address Mode Exceptions](#)
  - [Virtual 8086 Mode Exceptions](#)
- 

## FLD1/FLDL2T/FLDL2E/FLDPI/FLDLG2/FLDLN2/FLDZ-Load Constants

-----

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 E8	FLD1	X	X	X	X	X	X	Push +1.0 onto the FPU Stack
D9 E9	FLDL2T	X	X	X	X	X	X	Push log <sub>2</sub> 10 onto the FPU stack
D9 EA	FLDL2E	X	X	X	X	X	X	Push log <sub>2</sub> e onto the FPU stack
D9 EB	FLDPI	X	X	X	X	X	X	Load ð (pi) onto the FPU stack
D9 EC	FLDLG2	X	X	X	X	X	X	Push log <sub>2</sub> 102 onto the FPU stack

D9	ED	FLDLN2	X X X X X X Push log <sub>e</sub> 2 onto the FPU stack
D9	EE	FLDZ	X X X X X X Push +0.0 onto the FPU stack

## Description

Each of the constant instructions pushes a commonly-used constant (in extended-real format) onto the FPU stack.

## Operation

Decrement FPU stack-top pointer;  
ST(0) ← CONSTANT;

## Numeric Exceptions

IS

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

## Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

## Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

## Notes

ST(7) must be empty to avoid an invalid exception.

An internal 66-bit constant is used and rounded to external-real format (as specified by the RC bit of the control words). The precision exception is not raised.

---

# Related Information

- Description
- FPU Flags Affected
- Notes
- Operation
- Numeric Exceptions
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

---

# FLDCW-Load Control Word

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 /5	FLDCW <a href="#">m16</a>	<a href="#">X</a>	<a href="#">X</a>	<a href="#">X</a>	<a href="#">X</a>	<a href="#">X</a>	<a href="#">X</a>	Load FPU control word from <a href="#">m16</a>

---

## Description

FLDCW replaces the current value of the FPU control word with the value contained in the specified memory word.

---

## Operation

CW ← SRC;

---

## Numeric Exceptions

None, except for unmasking an existing exception.

---

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH; Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

FLDCW is typically used to establish or change the FPU's mode of operation.

If an exception bit in the status word is set, loading a new control word that unmask that exception will result in a floating-point error condition. When changing modes, the recommended procedure is to clear any pending exceptions before loading the new control word.

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## FLDENV-Load FPU Environment

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 /4	FLDENV <a href="#">m14byte/</a> <a href="#">m28byte</a>	X	X	X	X	X	X	Load FPU environment from m14byte or m28byte
D9 /4	FLDENW <a href="#">m14byte</a>				X	X	X	Load FPU environment from m14byte
D9 /4	FLDENVD <a href="#">m28byte</a>				X	X	X	Load FPU environment from m28byte

---

## Description

FLDENV reloads the FPU environment from the memory area defined by the source operand. This data should have been written by previous FSTENV or FNSTENV instruction.

The FPU environment consists of the FPU control word, status word, tag word, and error pointers (both data and instruction). The environment layout in memory depends on both the operand size and the current operating mode of the processor. The USE attribute of the current code segment determines the operand size: The 14-byte operand applies to a USE16 segment, and the 28-byte operand applies to a USE32 segment. Refer to the Intel documentation for figures of the environment layouts for both operand sizes in both real mode and protected mode. (In virtual-8086 mode, the real mode layout is used.) FLDENV should be run in the same operating mode as the corresponding FSTENV or FNSTENV.

---

## Operation

FPU environment ` SRC;

---

## Numeric Exceptions

None, except for loading an unmasked exception.

---

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

-----

=====

=====

---

=====

[illegible]

DC C8+i	FMUL ST(i),ST	X X X X X X Multiply ST(i) by ST
DE C8+i	FMULP ST(i),ST	X X X X X X Multiply ST(i) by ST and pop ST
DE /1	FIMUL m16int	X X X X X X Multiply ST by m16int
DA /1	FIMUL m32int	X X X X X X Multiply ST by m32int

-----

## Description

The multiplication instructions multiply the destination operand by the source operand and return the product to the destination.

-----

## Operation

DEST ← DEST \* SRC;  
IF instruction = FMULP THEN pop ST FI;

-----

## Numeric Exceptions

P, U, O, D, I.

-----

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH; Interrupt 7 if either EM or TS in CR0 is set.

-----

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Notes

If the source operand is in memory, it is automatically converted to the extended-real format.

## Related Information

- Description
- FPU Flags Affected
- Notes
- Operation
- Numeric Exceptions
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

## FNOP-No Operation

### Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 D0	FNOP	X	X	X	X	X	X	No operation is performed

### Description

FNOP performs no operation. It affects nothing except instruction pointers.

### Numeric Exceptions

None

### Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Related Information

- [Description](#)
- [FPU Flags Affected](#)
- [Numeric Exceptions](#)
- [Protected Mode Exceptions](#)
- [Real Address Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

---

## FPATAN-Partial Arctangent

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 F3	FPATAN	X	X	X	X	X	X	Replaces ST(1) with $\arctan(\text{ST}(1) \div \text{ST})$ and pop ST

---

## Description

The partial arctangent instruction computes the arctangent of  $\text{ST}(1) \div \text{ST}$ , and returns computed value, expressed in radians, to ST(1). It then

pops ST. The result has the same sign as the operand from ST(1), and a magnitude less than 0.

---

## Operation

```
ST(1) ← arctan(ST(1) ÷ ST);  
pop ST;
```

---

## Numeric Exceptions

P, U, D, I, IS.

---

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Notes

There is no restriction on the range of arguments that FPATAN can accept.

The fact that FPATAN takes two arguments and computes the arctangent of their ratio simplifies the calculation of other trigonometric functions. For instance, arcsin(x) (which is the arctangent of  $x \div \sqrt{1-x^2}$ ) can be computed using the following sequence of operations: Push x onto the FPU stack; compute  $\sqrt{1-x^2}$  and push the resulting value onto the stack; run FPATAN.

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

# FPREM-Partial Remainder

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 F8	FPREM	X	X	X	X	X	X	Replace ST with the remainder obtained on dividing ST by ST(1)

## Description

The partial remainder instruction computes the remainder obtained on dividing ST by ST(1), and leaves the result in ST. The sign of the remainder is the same as the sign of the original dividend in ST. The magnitude of the remainder is less than that of the modulus.

## Operation

```
EXPDIF ` exponent(ST) - exponent(ST(1));
IF EXPDIF < 64
THEN
  Q ` integer obtained by chopping ST ÷ ST(1) toward zero;
  ST ` ST - (ST(1) * Q);
  C2 ` 0;
  C0, C1, C3 ` three least-significant bits of Q; (* Q2, Q1, Q0 *)
ELSE
  C2 ` 1;
  N ` a number between 32 and 63;
  QQ ` integer obtained by chopping (ST ÷ ST(1)) ÷ 2EXPDIF-N
  toward zero;
  ST ` ST - (ST(1) * QQ * 2EXPDIF-N);
FI;
```

## Numeric Exceptions

U, D, I, IS.

---

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Notes

FREM produces an exact result; the precision (inexact) exception does not occur and the rounding control has no effect.

The FPREM instruction is not the remainder operation specified in IEEE Std 754. To get that remainder, the FPREM1 instruction should be used. FPREM is supported for compatibility with the 8087 and Intel287 math coprocessors.

FPREM works by iterative subtraction, can reduce the exponent of ST by no more than 63 in one execution. If FPREM succeeds in producing a remainder that is less than the modulus, the function is complete and the C2 flag is cleared. Otherwise, C2 is set, and the result in ST is called the *partial* remainder. The exponent of the partial remainder is less than the exponent of the original dividend by at least 32. Software can run the instruction again (using the partial remainder in ST as the dividend) until C2 is cleared. A higher-priority interrupting routine that needs the FPU can force a context switch between the instructions in the remainder loop.

An important use of FPREM is to reduce the arguments of periodic functions. When reduction is complete, FPREM provides the three least-significant bits of the quotient in flags C3, C1, and C0. This is important in argument reduction for the tangent function (using a modulus of  $\pi/4$ ), because it locates the original angle in the correct one of eight sectors of the unit circle.

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

# FPREM1-Partial Remainder

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 F5	FPREM1				X	X	X	Replace ST with the remainder obtained on dividing ST by ST(1)

---

## Description

The partial remainder instruction computes the remainder obtained on dividing ST by ST(1), and leaves the result in ST. The magnitude of the remainder is less than that of the modulus.

---

## Operation

```
EXPDIF ` exponent(ST) - exponent(ST(1));
IF EXPDIF < 64
THEN
  Q ` integer obtained by rounding ST ÷ ST(1) to the nearest integer;
      (*or the nearest even integer if the result is exactly halfway between 2 integers *)
  ST ` ST - (ST(1) * Q);
  C2 ` 0;
  Q ` ST - (ST(1) * Q);
  C0, C1, C3 ` three least-significant bits of Q; (* Q2, Q1, Q0 *)
  C2 ` 0;
  C0, C1, C3 ` three least-significant bits of Q; (* Q2, Q1, Q0 *)
ELSE
  C2 ` 1;
  N ` a number between 32 and 63;
  QQ ` integer obtained by chopping (ST ÷ ST(1)) ÷ 2EXPDIF-N;
      toward zero;
  ST ` ST - (ST(1) * QQ * 2EXPDIF-N);
FI;
```

---

## Numeric Exceptions

U, D, I, IS.

---

# Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

# Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

---

# Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Notes

FPREM1 produces an exact result; the precision (inexact) exception does not occur and the rounding control has no effect.

The FPREM1 instruction is not the remainder operation specified in IEEE Std 754. It differs from FPREM in the way it rounds the quotient of ST and ST(1), when the exponent difference of  $\text{exp}(\text{ST}) - \text{exp}(\text{ST})1$  is less than 64.

FPREM1 works by iterative subtraction, can reduce the exponent of ST by no more than 63 in one execution. If FPREM1 succeeds in producing a remainder that is less than one half the modulus, the function is complete and the C2 flag is cleared. Otherwise, C2 is set, and the result in ST is called the *partial* remainder. The exponent of the partial remainder is less than the exponent of the original dividend by at least 32. Software can run the instruction again (using the partial remainder in ST as the dividend) until C2 is cleared. A higher-priority interrupting routine that needs the FPU can force a context switch between the instructions in the remainder loop.

An important use of FPREM1 is to reduce the arguments of periodic functions. When reduction is complete, FPREM1 provides the three least-significant bits of the quotient in flags C3, C1, and C0. This is important in argument reduction for the tangent function (using a modulus of  $\pi/4$ ), because it locates the original angle in the correct one of eight sectors of the unit circle.

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## FPTAN-Partial Tangent

---

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 F2	FPTAN	X	X	X	X	X	X	Replace ST with its tangent and push 1 onto the FPU stack

---

# Description

The partial tangent instruction replaces the contents of ST with  $\tan(ST)$ , and then pushes 1.0 onto the FPU stack. ST, expressed in radians, must lie in the range  $|0| < 2^{63}$ .

---

# Operation

IF operand is in range  
THEN  
  C2 ← 0;  
  ST ←  $\tan(ST)$ ;  
  Decrement stack-top pointer;  
  ST ← 1.0;  
ELSE  
  C2 ← 1;  
FI;

---

# FPU Flags Affected

C0 C1 C2 C3  
? \* \* ?

C1, C2 as described in [FPU Flags Affected](#); C0, C3 undefined.

---

# Numeric Exceptions

P, U, D, I, IS.

---

# Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

# Real Address Mode Exceptions

Interrupt 7 either EM or TS in CR0 is set.

---

# Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Notes

If the operand is outside the acceptable range, the C2 flag is set, and ST remains unchanged. It is the programmer's responsibility to reduce the operand to an absolute value smaller than 2<sup>63</sup> by subtracting an appropriate integer multiple of 2<sup>0</sup>. Refer to the Intel documentation 6 for a discussion of the proper value to use for  $\hat{O}$  in performing such reductions.

The fact that FPTAN pushes 1.0 onto the FPU stack after computing tan(ST) maintains compatibility with the 8087 and Intel287 math coprocessors, and simplifies the calculation of other trigonometric functions. For instance, the cotangent (which is the reciprocal of the tangent) can be computed by running FDIVR after FPTAN.

ST(7) must be empty to avoid an invalid-operation exception.

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## FRNDINT-Round to Integer

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 FC	FRNDINT	X	X	X	X	X	X	Round ST to an integer

## Description

The round to integer instruction rounds the value in ST to an integer according to the RC field of the FPU control word.

## Operation

ST ← rounded ST;

## Numeric Exceptions

P, D, I, IS.

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

## Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

## Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

# Related Information

- Description
- FPU Flags Affected
- Operation
- Numeric Exceptions
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

## FRSTOR/FRSTRW/FRSTRD-Restore FRU State

### Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
DD /4	FRSTOR <a href="#">m94byte/</a> <a href="#">m108byte</a>	X	X	X	X	X	X	Load FPU state from m94byte or m108byte
DD /5	FRSTRW <a href="#">m94byte</a>			X	X	X		Load FPU state from m94byte
DD /4	FRSTRD <a href="#">m108byte</a>			X	X	X		Load FPU state from m108byte

### Description

FRSTOR reloads the FPU state (environment and register stack) from the memory area defined by the source operand. This data should have been written by a previous FSAVE or FNSAVE instruction.

The FPU environment consists of the FPU control word, status word, tag word, and error pointers (both data and instruction). The environment layout in memory depends on both the operand size and the current operating mode of the processor. The USE attribute of the current code segment determines the operand size: the 14-byte operand applies to a USE16 segment, and the 28-byte operand applies to a USE32 segment. Refer to the Intel documentation for the environment layouts for both operand sizes in both real mode and protected mode. (In virtual-8086 mode, the real mode layout is used.) The stack registers, beginning with ST and ending with ST(7), are in the 80 bytes that immediately follow the environment image. FRSTOR should be run in the same operating mode as the corresponding FSAVE or FNSAVE.

### Operation

FPU state ← SRC;

---

## Numeric Exceptions

None, except for loading an unmasked exception.

---

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH; Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

If the state image contains an unmasked exception, loading it will result in a floating-point error condition.

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

# FSAVE/FNSAVE-Store FPU State

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
9B DD /6	FSAVE m94byte/ m108byte	X	X	X	X	X	X	Store FPU state to m94byte or m108byte after checking for unmasked FP error condition; then re-initialize the CPU
9B DD /6	FSAVEW m94byte			X	X	X		Store FPU state to m94byte after checking for unmasked FP error condition; then re-initialize the FPU
9B DD /6	FSAVED m108byte			X	X	X		Store FPU state to m108byte after checking for unmasked FP error condition; then re-initialize the FPU
DD /6	FNSAVE m94byte/ m108byte	X	X	X	X	X	X	Store FPU environment to m94byte or m108byte without checking for unmasked FP error condition; then re-initialize the FPU
DD /6	FNSAVEW m94byte			X	X	X		Store FPU environment to m94byte without checking for unmasked FP error condition; then re-initialize the FPU
DD /6	FNSAVED m108byte			X	X	X		Store FPU environment to m108byte without checking for unmasked FP error condition; then re-initialize the FPU

## Description

The save instructions write the current FPU state (environment and register stack) to the specified destination, and then re-initialize the FPU. The environment consists of the FPU control word, status word, tag word, and error pointers (both data and instruction).

The state layout in memory depends on both operand size and the current operating mode of the processor. The USE attribute of the current code segment determines the operand size: the 94-byte operand applies to USE16 segment, and the 108-byte operand applies to a USE32 segment. Refer to the Intel documentation for the environment layouts for both operand sizes in both real mode and protected mode. (In virtual-8086 mode, the real mode layout is used.) The stack registers, beginning with ST and ending with ST(7), are stored in the 80 bytes that immediately follow the environment image.

## Operation

DEST ` FPU state;  
initialize FPU; (\* Equivalent to FNINIT \*)

-----

## FPU Flags Affected

C0	C1	C2	C3
0	0	0	0

C0, C1, C2, C3 cleared.

-----

## Numeric Exceptions

None

-----

## Protected Mode Exceptions

#GP(0) if the destination is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH; Interrupt 7 if either EM or TS in CR0 is set.

-----

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Notes

FSAVE and FNSAVE do not store the FPU state until all FPU activity is complete. Thus, the saved image reflects the state of the FPU after any previously decoded instruction has been run.

If a program is to read from the memory image of the state following a save instruction, it must issue an FWAIT instruction to ensure that the storage is complete.

The save instructions are typically used when an operating system needs to perform a context switch, or an exception handler needs to use the FPU, or an application program wants to pass a "clean" FPU to a subroutine.

---

## Related Information

- [Description](#)
- [FPU Flags Affected](#)
- [Notes](#)
- [Operation](#)
- [Numeric Exceptions](#)
- [Protected Mode Exceptions](#)
- [Real Address Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

---

## FSCALE-Scale

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 FD	FSCALE	X	X	X	X	X	X	Scale ST by ST(1)

---

## Description

The scale instruction interprets the value in (ST(1)) as an integer, and adds this integer to the exponent of ST. Thus, FSCALE provides rapid multiplication or division by integral powers of 2.

---

## Operation

ST ← ST \* 2<sup>ST(1)</sup>;

---

## Numeric Exceptions

P, U, O, D, I, IS.

---

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Notes

FSCALE can be used as an inverse to FXTRACT. Because FSCALE does not pop the exponent part, however, FSCALE must be followed by FSTP ST(1) in order to completely undo the effect of a preceding FXTRACT.

There is no limit on the range of the scale factor in ST(1). If the value is not integral, FSCALE uses the nearest integer smaller in magnitude; i.e., it chops the value toward 0. If the resulting integer is zero, the value in ST is not changed.

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

---

# FSETPM-Set Protected Mode

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
DB E4	FSETPM			X				Sets the operating mode of the 80287 to protected virtual address mode

---

## Description

Sets the operating mode of the 80287 to Protected Virtual-Address mode. When the 80287 is first initialized following hardware RESET, it operates in Real-Address mode, just as does the 80286 CPU. Once the 80287 NPX has been set into Protected mode, only a hardware RESET can return the NPX to operation in Real-Address mode.

When the 80287 operates in Protected mode, the NPX exception pointers are represented differently than they are in Real-Address mode (see the FSAVE and FSTENV instructions). This distinction is evident primarily to writers of numeric exception handlers, however. For general application programmers, the operating mode of the 80287 need not be a concern.

---

## Operation

---

## FPU Flags Affected

C0 C1 C2 C3

---

## Numeric Exceptions

---

## Protected Mode Exceptions

---

## Real Address Mode Exceptions

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

---

## FSIN-Sine

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 FE	FSIN							<a href="#">X</a> <a href="#">X</a> <a href="#">X</a> Replace ST with its sine

---

## Description

The sine instruction replaces the contents of ST with  $\sin(\text{ST})$ . ST, expressed in radians, must lie in the range  $|\text{ST}| < 2^{\text{63}}$ .

---

## Operation

If operand is in range  
THEN  
  C2 ← 0;  
ELSE  
  C2 ← 1;  
FI;

---

## FPU Flags Affected

C0	C1	C2	C3
?	*	*	?

C1, C2 as described in [FPU Flags Affected](#); C0, C3 undefined.

---

## Numeric Exceptions

P, U, D, I, IS.

---

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Notes

If the operand is outside the acceptable range, the C2 flag is set, and ST remains unchanged. It is the programmer's responsibility to reduce the operand to an absolute value smaller than 2<sup>63</sup> by subtracting an appropriate integer multiple of 2<sup>0</sup>. Refer to the Intel documentation for a discussion of the proper value to use the  $\pi$  in performing such reductions.

# Related Information

- Description
- FPU Flags Affected
- Notes
- Operation
- Numeric Exceptions
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

# FSINCOS-Sine and Cosine

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 FB	FSINCOS	X	X	X				Compute the sine and cosine of ST; replace ST with the sine, then push the cosine onto the FPU stack

# Description

FSINCOS computes both sin(ST) and cos(ST), replaces ST with the sine and then pushes the cosine onto the FPU stack. ST, expressed in radians, must lie in the range  $|0| < 2^{63}$ .

# Operation

IF operand is in range

```

THEN
  C2 ` 0;
  TEMP ` cos(ST);
  ST ` sin(ST);
  Decrement FPU stack-top pointer;
  ST ` TEMP;
ELSE
  C2 ` 1;
FI:

```

---

## FPU Flags Affected

C0 C1 C2 C3

? \* \* ?

C1, C2 as described in [FPU Flags Affected](#); C0, C3 undefined.

---

## Numeric Exceptions

P, U, D, I, IS.

---

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Notes

If the operand is outside the acceptable range, the C2 flag is set, and ST remains unchanged. It is the programmer's responsibility to reduce the operand to an absolute value smaller than 2<sup>63</sup> by subtracting an appropriate integer multiple of 2<sup>0</sup>. Refer to the Intel documentation for a discussion of the proper value to use for  $\bar{O}$  in performing such reductions.

It is faster to run FSINCOS than to run both FSIN and FCOS.

---

## Related Information

- [Description](#)
  - [FPU Flags Affected](#)
  - [Notes](#)
  - [Operation](#)
  - [Numeric Exceptions](#)
  - [Protected Mode Exceptions](#)
  - [Real Address Mode Exceptions](#)
  - [Virtual 8086 Mode Exceptions](#)
- 

## FSQRT-Square Root

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 FA	FSQRT	X	X	X	X	X	X	Replace ST with its square root

---

## Description

The square root instruction replaces the value in ST with its square root.

---

## Operation

ST ← square root of ST;

---

## Numeric Exceptions

P, D, I, IS.

---

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Notes

The square root of -0 is -0.

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## FST/FSTP-Store Real

---

## Details Table

Encoding	Instruction	0 1 2 3 4 5	Description
D9 /2	FST <i>m32real</i>	X X X X X X	Copy ST to m32real
DD /2	FST <i>m64real</i>	X X X X X X	Copy ST to m64real
DD D0+i	FST <i>ST(i)</i>	X X X X X X	Copy ST to ST(i)
D9 /3	FSTP <i>m32real</i>	X X X X X X	Copy ST to m32real and pop ST
DD /3	FSTP <i>m64real</i>	X X X X X X	Copy ST to m64real and pop ST
DB /7	FSTP <i>m80real</i>	X X X X X X	Copy ST to m80real and pop ST
DD D8+i	FSTP <i>ST(i)</i>	X X X X X X	Copy ST to ST(i) and pop ST

## Description

FST copies the current value in the ST register to the destination, which can be another register or a single- or double-real memory operand. FSTP copies and then pops ST; it accepts extended-real memory operands as well as the types accepted by FST.

If the source is register, the register number used is that before the stack is popped.

## Operation

DEST ← ST(0);  
IF instruction = FSTP THEN pop ST FI;

## Numeric Exceptions

Register or extended-real destinations: IS  
Single- or double-real destinations: P, U, O, I, IS

## Protected Mode Exceptions

#GP(0) if the destination is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in SS segment; #PF(fault-code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH; Interrupt 7 if either EM or TS in CR0 is set.

# Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Notes

If the destination is single- or double-real, the significand is rounded to the width of the destination according to the RC field of the control word, and the exponent is converted to the width and bias of the destination format. The over/underflow condition is checked for as well.

If ST contains zero,  $\pm\infty$ , or a NaN, then the significand is not rounded, but chopped (on the right) to fit the destination. Nor is the exponent converted; it too is chopped on the right. These operations preserve the value's identity as  $\infty$  or NaN (exponent all ones).

The invalid-operation exception is not raised when the destination is a nonempty stack element.

A denormal operand in ST(0) causes an underflow. No denormal operand exception is raised.

## Related Information

- [Description](#)
- [FPU Flags Affected](#)
- [Notes](#)
- [Operation](#)
- [Numeric Exceptions](#)
- [Protected Mode Exceptions](#)
- [Real Address Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

## FSTCW/FNSTCW-Store Control Word

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
----------	-------------	---	---	---	---	---	---	-------------

9B D9 /7	FSTCW m16	X X X X X X	Store FPU control word to m16 after checking for unmasked FP error condition
D9 /7	FNSTCW m16	X X X X X X	Store FPU control word to m16 without checking for unmasked FP error condition

## Description

FSTCW and FNSTCW write the current value of the FPU control word to the specified destination.

## Operation

DEST ← CW;

## Numeric Exceptions

None

## Protected Mode Exceptions

#GP(0) if the destination is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH; Interrupt 7 if either EM or TS in CR0 is set.

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Notes

FSTCW checks for unmasked floating-point error conditions before storing the control word FNSTCW does not.

## Related Information

- Description
- FPU Flags Affected
- Notes
- Operation
- Numeric Exceptions
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

## FSTENV/FNSTENV-Store FPU Environment

### Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
9B D9 /6	FSTENV m14byte/ m28byte	X	X	X	X	X	X	Store FPU environment to m14byte or m28byte after checking for unmasked FP error condition; then mask all FP exceptions.
9B D9 /6	FSTENVW m14byte		X	X	X			Store FPU environment to m14byte after checking for unmasked FP error condition; then mask all FP exceptions
9B D9 /6	FSTENV D m28byte		X	X	X			Store FPU environment to m28byte after checking for unmasked FP error condition; then mask all FP exceptions
D9 /6	FNSTENV m14byte/ m28byte	X	X	X	X	X	X	Store FPU environment to m14byte or m28byte without checking for unmasked FP error condition; then mask all FP exceptions
D9 /6	FNSTENVW m14byte		X	X	X			Store FPU environment to m14byte without checking for unmasked FP error condition; then mask all FP exceptions
D9 /6	FNSTENV D m28byte		X	X	X			Store FPU environment to m28byte without checking for unmasked FP error condition; then mask all FP exceptions

---

## Description

The store environment instructions write the current FPU environment to the specified destination, and then mask all floating-point exceptions. The FPU environment consists of the FPU control word, status word, tag word, and error pointer (both data and instruction).

The environment layout in memory depends on both the operand size and the current operating mode of the processor. The USE attribute of the current code segment determines the operand size: the 14-byte operand applies to a USE16 segment, and the 28-byte operand applies to a USE32 segment. Figures 6-6 through 6-8 show the environment layouts for both operand sizes in both real mode and protected mode. (In virtual-8086 mode, the real mode layout is used).

---

## Operation

DEST` FPU environment;  
CW[0..5]` 111111B;

---

## Numeric Exceptions

None

---

## Protected Mode Exceptions

#GP(0) if the destination is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH; Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

FSTENV and FNSTENV do not store the environment until FPU activity is complete. Thus, the save environment reflects the state of the FPU after any previously decoded instruction has been run.

The store environment instructions are often used by exception handlers because they provide access to the FPU error pointers. The environment is typically saved onto the memory stack. After saving the environment, FSTENV and FNSTENV sets all the exception masks in the FPU control word. This prevents floating-point errors from interrupting the exception handler.

FSTENV checks for unmasked floating-point error conditions before storing the FPU environment; FNSTENV does not.

## Related Information

- Description
- FPU Flags Affected
- Notes
- Operation
- Numeric Exceptions
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

## FSTSW/FNSTSW-Store Status Word

### Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
9B DF /7	FSTSW m16	X	X	X	X	X	X	Store FPU status word to m16 after checking for unmasked FP error condition
9B DF E0	FSTSW AX		X	X	X	X		Store FPU status word to AX register after checking for unmasked FP error condition
DF /7	FNSTSW m16	X	X	X	X	X	X	Store FPU status word to m16 without checking for unmasked FP error condition
DF E0	FNSTSW AX		X	X	X	X		Store FPU status word to AX register without checking for unmasked FP error condition

### Description

FSTSW and FNSTSW write the current value of the FPU status word to the specified destination, which can be either a two-byte location in memory or the AX register.

---

## Operation

DEST ` SW;

---

## Numeric Exceptions

None

---

## Protected Mode Exceptions

#GP(0) if the destination is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in SS segment; #PF(fault-code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH; Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

FSTSW checks for unmasked floating-point error conditions before storing the status word; FNSTSW does not.

FSTSW and FNSTSW are used primarily in conditional branching (after a comparison, FPREM, FPREM1, or FXAM instruction). They can also be used to invoke exception handlers (by polling the exception bits) in environments that do not use interrupts.

When FNSTSW AX is run, the AX register is updated before the processor runs any further instructions. The status stored is that from the completion of the prior ESC instruction.

---

## Related Information

- Description
- FPU Flags Affected
- Notes
- Operation
- Numeric Exceptions
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

# FSUB/FSUBP/FISUB-Subtract

## Details Table

Encoding	Instruction	0 1 2 3 4 5	Description
D8 /4	FSUB <a href="#">m32real</a>	X X X X X X	Replace ST with ST - m32real
DC /4	FSUB <a href="#">m64real</a>	X X X X X X	Replace ST with ST - m64real
D8 E0+i	FSUB ST, <a href="#">ST(i)</a>	X X X X X X	Replace ST with ST - ST(i)
DC E8+i	FSUB <a href="#">ST(i)</a> , ST	X X X X X X	Replace ST(i) with ST(i) - ST
DE E8+i	FSUBP <a href="#">ST(i)</a> , ST	X X X X X X	Replace ST(i) with ST - ST(i); pop ST
DE E9	FSUB	X X X X X X	Replace ST(1) with ST - ST(1); pop ST
DE /4	FISUB <a href="#">m16int</a>	X X X X X X	Replace ST with ST - m16int
DA /4	FISUB <a href="#">m32int</a>	X X X X X X	Replace ST with ST - m32int

## Description

The subtraction instructions subtract the other operand from the stack top and return the difference to the destination.

## Operation

DEST`ST - Other Operand;  
IF instruction = FSUBP THEN pop ST FI;

---

## Numeric Exceptions

P, U, O, D, I, IS.

---

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in SS segment; #PF(fault-code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH; Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

If the source operand is in memory, it is automatically converted to the extended-real format.

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

# FSUBR/FSUBRP/FISUBR-Reverse Subtract

---

## Details Table

Encoding	Instruction	0 1 2 3 4 5	Description
D8 /5	FSUBR <i>m32real</i>	X X X X X X	Replace ST with <i>m32real</i> - ST
DC /5	FSUBR <i>m64real</i>	X X X X X X	Replace ST with <i>m64real</i> - ST
D8 E8+i	FSUBR ST, <i>ST(i)</i>	X X X X X X	Replace ST with <i>ST(i)</i> - ST
DC E0+i	FSUBR <i>ST(i)</i> , ST	X X X X X X	Replace <i>ST(i)</i> with ST - <i>ST(i)</i>
DE E0+i	FSUBRP <i>ST(i)</i> , ST	X X X X X X	Replace <i>ST(i)</i> with ST - <i>ST(i)</i>
DE E1	FSUBR	X X X X X X	Replace ST(1) with ST - ST(1); pop ST
DE /5	FISUBR <i>m16int</i>	X X X X X X	Replace ST with <i>m16int</i> - ST
DA /5	FISUBR <i>m32int</i>	X X X X X X	Replace ST with <i>m32int</i> - ST

---

## Description

The reverse subtraction instructions subtract the stack top from the other operand and return the difference to the destination.

---

## Operation

DEST ` Other Operand - ST;  
IF instruction = FSUBRP THEN pop ST FI;

---

## Numeric Exceptions

P, U, O, D, I, IS.

---

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in SS segment; #PF(fault-code) for a page fault; #NM if either EM or TS in CR0 is set; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH; Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

If the source operand is in memory, it is automatically converted to the extended-real format.

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## FTST-Test

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 E4	FTST	X	X	X	X	X	X	Compare ST with 0.0

## Description

The test instruction compares the stack top to 0.0. Following the instruction, the condition codes reflect the result of the comparison.

## Operation

CASE (relation of operands) OF  
 Not comparable: C3, C2, C0 ` 111;  
 ST > SRC: C3, C2, C0 ` 000;  
 ST < SRC: C3, C2, C0 ` 001;  
 ST = SRC: C3, C2, C0 ` 100;

## FPU Flags Affected

FPU FLAGS	EFLAGS
C0	CF
C1	(none)
C2	PF
C3	ZF

C1 as described in [FPU Flags Affected](#); C0, C2, C3 as specified above.

## Numeric Exceptions

D, I, IS.

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

# Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

# Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

## Notes

If ST contains a NaN or an object of undefined format, or if a stack fault occurs, the invalid-operation exception is raised, and the condition bits are set to "unordered."

The sign of zero is ignored, so that -0.0=+0.0.

## Related Information

- Description
- FPU Flags Affected
- Notes
- Operation
- Numeric Exceptions
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

# FUCOM/FUCOMP/FUCOMPP-Unordered Compared Real

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
DD E1	FUCOM				X	X	X	Compare ST with ST(1)
DD E0+i	FUCOM <a href="#">ST(i)</a>				X	X	X	Compare ST with ST(i)

DD E9	FUCOMP	X X X Compare ST with ST(1) and pop ST
DD E8+i	FUCOMP ST(i)	X X X Compare ST with ST(i) and pop ST
DA E9	FUCOMPP	X X X Compare ST with ST(1) and pop ST twice

-----

## Description

The unordered compare real instructions compare the stack top to the source, which must be a register. If no operand is encoded, ST is compared to ST(1). Following the instruction, the condition codes reflect the relation between ST and the source operand.

-----

## Operation

CASE (relation of operands) OF

Not comparable: C3, C2, C0`111;

ST > SRC: C3, C2, C0`000;

ST < SRC: C3, C2, C0`001;

ST = SRC: C3, C2, C0`100;

IF instruction = FUCOMP THEN pop ST; FI;

IF instruction = FUCOMPP THEN pop ST; pop ST; FI;

-----

## FPU Flags Affected

FPU FLAGS	EFLAGS
C0	CF
C1	(none)
C2	PF
C3	ZF

C1 as described in [FPU Flags Affected](#); C0, C2, C3 as specified above.

-----

## Numeric Exceptions

D, I, IS.

-----

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

# Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

---

# Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

# Notes

If either operand is an SNaN or is in an undefined format, or if a stack fault occurs, the invalid-operation exception is raised, and the condition bits are set to "unordered."

If either operand is a QNaN, the condition bits are set to "unordered." Unlike the ordinary compare instructions (FCOM, etc.), the unordered compare instructions do not raise the invalid-operation exception on account of a QNaN operand.

The sign of zero is ignored, so that  $-0.0=+0.0$ .

---

# Related Information

- [Description](#)
- [FPU Flags Affected](#)
- [Notes](#)
- [Operation](#)
- [Numeric Exceptions](#)
- [Protected Mode Exceptions](#)
- [Real Address Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

---

# FWAIT-Wait

---

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
9B	FWAIT	X	X	X	X	X	X	Alias for WAIT

## Description

FWAIT causes the processor to check for pending unmasked numeric exceptions before proceeding.

## Numeric Exceptions

None

## Protected Mode Exceptions

#NM if both MP and TS in CR0 are set.

## Real Address Mode Exceptions

Interrupt 7 if both MP and TS in CR0 are set.

## Virtual 8086 Mode Exceptions

#NM if both MP and TS in CR0 are set.

## Notes

As its opcode shows, FWAIT is not actually an ESC instruction, but an alternate mnemonic for WAIT.

Coding FWAIT after an ESC instruction ensures that any unmasked floating-point exceptions the instruction may cause are handled before the processor has a chance to modify the instruction's results.

Refer to the Intel documentation for more information about when to use FWAIT.

## Related Information

- Description
- FPU Flags Affected
- Notes
- Numeric Exceptions
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

# FXAM-Examine

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 E5	FXAM	X	X	X	X	X	X	Report the type of object in the ST register

## Description

The examine instruction reports the type of object contained in the ST register by setting the FPU Flags.

## Operation

C1 ` sign bit of ST; (\* 0 for positive, 1 for negative \*)

CASE (type of object in ST) OF  
  Unsupported: C3, C2, C0 ` 000;  
  NaN: C3, C2, C0 ` 001;  
  Normal: C3, C2, C0 ` 010;  
  Infinity: C3, C2, C0 ` 011;  
  Zero: C3, C2, C0 ` 100;  
  Empty: C3, C2, C0 ` 101;  
  Denormal: C3, C2, C0 ` 110;

## FPU Flags Affected

FPU FLAGS	EFLAGS
C0	CF
C1	(none)
C2	PF
C3	ZF

C0, C1, C2, C3 as shown above.

## Numeric Exceptions

None

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

## Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

## Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

## Notes

C1 bit represents the sign of ST(0) regardless of whether ST(0) is empty or full.

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

# FXCH-Exchange Register Contents

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 C9	FXCH	X	X	X	X	X	X	Exchange the contents of ST and ST(1)
D9 C8+i	FXCH ST(i)	X	X	X	X	X	X	Exchange the contents of ST and ST(i)

## Description

FXCH swaps the contents of the destination and stack-top registers. If the destination is not coded explicitly, ST(1) is used.

## Operation

TEMP`ST;  
ST`DEST;  
DEST`TEMP;

## Numeric Exceptions

IS.

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Notes

Many numeric instructions operate only on the stack top; FXCH provides a simple means for using these instructions on lower stack elements. For example, the following sequence takes the square root of the third register from the top (assuming that ST is nonempty):

```
FXCH ST (3)
```

```
FSQRT
```

```
FXCH ST (3)
```

FXCH can be paired with some floating point instructions (i.e., FADD, FSUB, FMUL, FLD, FCOM, FUCOM, FCHS, FTST, FABS, FDIV. This set also includes the FADDP, FSUBRP, etc., instructions). When paired, the FXCH gets run in parallel, and does not take any additional clocks.

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## FXTRACT-Extract Exponent and Significand

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 F4	FXTRACT	X	X	X	X	X	X	Separate ST into its exponent and significand; replace ST with exponent then push significand onto FPU stack

---

## Description

FXTRACT splits the value in ST into its exponent and significand. The exponent replaces the original operand on the stack and the significand is pushed onto the stack. Following execution of FXTRACT, ST (the new stack top) contains the value of the original significand expressed as a real number: its sign is the same as the operand's, its exponent is 0 true (16,383 or 3FFFH biased), and its significand is identical to the original operand's. ST(1) contains the value of the original operand's true (unbiased) exponent expressed as a real number.

To illustrate the operation of FXTRACT, assume that ST contains a number whose true exponent is +4 (i.e., its exponent field contains 4003H). After running FXTRACT, ST(1) will contain the real number +4.0; its sign will be positive, its exponent field will contain 4001H (+2 true) and its significand field will contain 1000...00B. In other words, the value in ST(1) will be  $1.0 * 2^2 = 4$ . If ST contains an operand whose true exponent is -7 (i.e., its exponent field contains 3FF8H), then FXTRACT will return an "exponent" of -7.0; after the instruction runs, ST(1)'s sign and exponent fields will contain C001H (negative sign, true exponent of 2), and its significand will be 1001100...00B. In other words, the value in ST(1) will be  $-1.75 * 2^2 = -7.0$ . In both cases, following FXTRACT, ST's sign and significand fields will be the same as the original operand's, and its exponent field will contain 3FFFH (0 true).

---

## Operation

TEMP ← significand of ST;  
ST ← exponent of ST;  
Decrement FPU stack-top pointer;  
ST ← TEMP;

---

## Numeric Exceptions

Z, D, I, IS.

---

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Notes

FXTRACT (extract exponent and significand) performs a superset of the IEEE-recommended **logb(*x*)** function.

If the original operand is zero, FXTRACT leaves - $\hat{y}$  in ST(1) (the exponent) while ST is assigned the value zero with a sign equal to that of the original operand. The zero-divide exception is raised in this case, as well.

ST(7) must be empty to avoid the invalid-operation exception.

FXTRACT is useful for power and range scaling operations. Both FXTRACT and the base 2 exponential instruction F2XM1 are needed to perform a general power operation. Converting numbers in extended-real format to decimal representations (e.g., for printing or displaying) requires not only FBSTP but also FXTRACT to allow scaling that does not overflow the range of the extended format. FXTRACT can also be useful for debugging, because it allows the exponent and significand parts of a real number to be examined separately.

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## FYL2X-Compute $y * \log_2 x$

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 F1	FYL2X	X	X	X	X	X	X	Replace ST with $ST(1) * \log_2 ST$ and pop ST

## Description

FYL2X computes the base-2 logarithm of ST, multiplies the logarithm by ST(1), and returns the resulting value to ST(1). It then pops ST. The operand in ST must not be negative or zero.

## Operation

$ST(1) \leftarrow ST(1) * \log_2 ST;$   
pop ST;

## Numeric Exceptions

P, U, O, Z, D, I, IS.

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

## Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

## Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

## Notes

If the operand in ST is negative, the invalid-operation exception is raised.

The FYL2X instruction is designed with a built-in multiplication to optimize the calculation of logarithms with arbitrary positive base:

$$\log_b x = (\log_2 x) \cdot \log_2 b$$

The instructions FLDL2T and FLDL2E load the constants  $\log_2 10$  and  $\log_2 e$ , respectively.

## Related Information

- Description
- FPU Flags Affected
- Notes
- Operation
- Numeric Exceptions
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

## FYL2XP1-Compute $y \cdot \log_2(x + 1)$

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D9 F9	FYL2XP1	X	X	X	X	X	X	Replace ST(1) with ST(1) * $\log_2(ST+1.0)$ and pop ST

## Description

FYL2XP1 computes the base-2 logarithm of (ST+1.0), multiplies the logarithm by ST(1), and returns the resulting value to ST(1). It then pops ST. The operand in ST must be in the range

$$-(1-(1/2)) \leq ST \leq 1/2$$

## Operation

```
ST(1) ` ST(1) * log2(ST+1.0);  
pop ST;
```

---

## Numeric Exceptions

P, U, D, I, IS.

---

## Protected Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Real Address Mode Exceptions

Interrupt 7 if either EM or TS in CR0 is set.

---

## Virtual 8086 Mode Exceptions

#NM if either EM or TS in CR0 is set.

---

## Notes

If the operand in ST is outside the acceptable range, the result of FYL2XP1 is undefined.

The FYL2XP1 instruction provides improved accuracy over FYL2X when computing the logarithms of numbers very close to 1. When  $\epsilon$  is small, more significant digits can be retained by providing  $\epsilon$  as an argument to FYL2XP1 than by providing  $1+\epsilon$  as an argument to FYL2X.

---

## Related Information

[Description](#)

[FPU Flags Affected](#)

[Notes](#)

[Operation](#)

[Numeric Exceptions](#)

[Protected Mode Exceptions](#)

# HLT-Halt

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
F4	HLT	X	X	X	X	X	X	Halt

## Description

The HLT instruction stops instruction processing and places the processor in a HALT state. An enabled interrupt, NMI, or a reset will resume execution. If an interrupt (including NMI) is used to resume execution after a HLT instruction, the saved CS:IP (or CS:EIP) value points to the instruction following the HLT instruction.

## Operation

Enter Halt state;

## Protected Mode Exceptions

The HLT instruction is a privileged instruction; #GP(0) if the current privilege level is not 0.

## Virtual 8086 Mode Exceptions

#GP(0); the HLT instruction is a privileged instruction.

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Protected Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

# IDIV-Signed Divide

## Details Table

Encoding	Instruction	0 1 2 3 4 5	Description
F6 /7	IDIV <i>r/m8</i>	X X X X X X	Signed divide AX (where AH must contain sign-extension of AL) by <i>r/m</i> byte (results: AL=Quotient, AH=Remainder)
F7 /7	IDIV <i>r/m16</i>	X X X X X X	Signed divide DX:AX (where DX must contain sign-extension of AX) by <i>r/m</i> word (results: AX=Quotient, DX=Remainder)
F7 /7	IDIV <i>r/m32</i>	X X X	Signed divide EDX:EAX (where EDX must contain sign-extension of EAX) by <i>r/m</i> dword (results: EAX=Quotient, EDX=Remainder)

## Description

The IDIV instruction performs a signed division. The dividend, quotient, and remainder are implicitly allocated to fixed registers. Only the divisor is given as an explicit *r/m* operand. The type of the divisor determines which registers to use as follows:

SIZE	DIVIDEND	DIVISOR	QUOTIENT	REMAINDER
byte	AX	<i>r/m8</i>	AL	AH
word	DX:AX	<i>r/m16</i>	AX	DX
dword	EDX:EAX	<i>r/m32</i>	EAX	EDX

If the resulting quotient is too large to fit in the destination, or if the divisor is 0, an Interrupt 0 is generated. Nonintegral quotients are truncated toward 0. The remainder has the same sign as the dividend and the absolute value of the remainder is always less than the absolute value of the divisor.

---

## Operation

```
temp ← dividend / divisor;
IF temp does not fit in quotient
THEN Interrupt 0;
ELSE
  quotient ← temp;
  remainder ← dividend MOD (r/m);
FI;
```

**Note:** Divisions are signed. The dividend must be sign-extended. The divisor is given by the *r/m* operand. The dividend, quotient, and remainder use implicit registers. Refer to the table under "Description."

---

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
?			?	?	?	?	?

The OF, SF, ZF, AF, PF, and CF flags are undefined.

---

## Protected Mode Exceptions

Interrupt 0 if the quotient is too large to fit in the designated register (AL or AX), or if the divisor is 0; #GP (0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 0 if the quotient is too large to fit in the designated register (AL or AX), or if the divisor is 0; Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Related Information

[Description](#)

- Flags Affected
- Operation
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

# IMUL-Signed Multiply

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
F6 /5	IMUL <a href="#">r/m8</a>	X	X	X	X	X	X	AX ` AL * r/m byte
F7 /5	IMUL <a href="#">r/m16</a>	X	X	X	X	X	X	DX:AX ` AX * r/m word
F7 /5	IMUL <a href="#">r/m32</a>				X	X	X	EDX:EAX ` EAX * r/m dword
0F AF /r	IMUL <a href="#">r16,r/m16</a>				X	X	X	r16 ` r16 * r/m word
0F AF /r	IMUL <a href="#">r32,r/m32</a>				X	X	X	r32 ` r32 * r/m dword
6B /r ib	IMUL <a href="#">r16,r/m16,imm8</a>	X	X	X	X	X	X	r16 ` r/m word * sign-extended immediate byte
6B /r ib	IMUL <a href="#">r32,r/m32,imm8</a>				X	X	X	r32 ` r/m dword * sign-extended immediate byte
6B /r ib	IMUL <a href="#">r16,imm8</a>	X	X	X	X	X	X	r16 ` r16 * sign-extended immediate byte
6B /r ib	IMUL <a href="#">r32,imm8</a>				X	X	X	r32 ` r32 * sign-extended immediate byte
69 /r iw	IMUL <a href="#">r16,r/m16,imm16</a>	X	X	X	X	X	X	r16 ` r/m word * immediate word
69 /r id	IMUL <a href="#">r32,r/m32,imm32</a>				X	X	X	r32 ` r/m dword * immediate dword
69 /r iw	IMUL <a href="#">r16,imm16</a>	X	X	X	X	X	X	r16 ` r16 * immediate word
69 /r id	IMUL <a href="#">r32,imm32</a>				X	X	X	r32 ` r32 * immediate dword

## Description

The IMUL instruction performs signed multiplication. Some forms of the instruction use implicit register operands. The operand combinations for all forms of the instruction are shown in the "Description" column above.

The IMUL instruction clears the OF and CF flags under the following conditions (otherwise the CF and OF flags are set):

INSTRUCTION FORM CONDITION FOR CLEARING CR AND OF

r/m8	AL = sign-extend of AL to 16-bits
r/m16	AX = sign-extend of AX to 32-bits
r/m32	EDX:EAX = sign-extend of EAX to 32-bits
r16,r/m16	Result exactly fits within r16
r32,r/m32	Result exactly fits within r32
r16,r/m16,imm16	Result exactly fits within r16
r32,r/m32,imm32	Result exactly fits within r32

# Operation

result ` multiplicand \* multiplier;

# Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
*		?	?	?	?	*	

The OF and CF flags as described in the table in the "Description" section above; the SF, ZF, AF, and PF flags are undefined.

# Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

# Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

# Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

-----

# Notes

When using the accumulator forms (IMUL **r/m8**, IMUL **rm16**, or IMUL **r/m32**), the result of the multiplication is available even if the overflow flag is set because the result is twice the size of the multiplicand and multiplier. This is large enough to handle any possible result.

-----

# Related Information

- Description
  - Flags Affected
  - Notes
  - Operation
  - Protected Mode Exceptions
  - Real Address Mode Exceptions
  - Virtual 8086 Mode Exceptions
- 

# IN-Input from Port

-----

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
E4 ib	IN AL,imm8	X	X	X	X	X	X	Input byte from port imm8 into AL
E5 ib	IN AX,imm8	X	X	X	X	X	X	Input word from port imm8 into AX
E5 ib	IN EAX,imm8				X	X	X	Input dword from port imm8 into EAX
EC	IN AL,DX	X	X	X	X	X	X	Input byte from port DX into AL
ED	IN AX,DX	X	X	X	X	X	X	Input word from port DX into AX
ED	IN EAX,DX				X	X	X	Input dword from port DX into EAX

-----

# Description

The IN instruction transfers a data byte or data word from the port numbered by the second operand into the register (AL, AX, or EAX)

specified by the first operand. Access any port from 0 to 65535 by placing the port number in the DX register and using an IN instruction with the DX register as the second parameter. These I/O instructions can be shortened by using an 8-bit port I/O in the instruction. The upper eight bits of the port address will be 0 when 8-bit port I/O is used.

---

## Operation

```
IF (PE = 1) AND ((VM = 1) OR (CPL > IOPL))
THEN (* Virtual 8086 mode, or protected mode with CPL > IOPL *)
  IF NOT I-O-Permission (SRC, width(SRC))
  THEN #GP(0);
  FI;
FI;
DEST ` [SRC]; (* Reads from I/O address space *)
```

---

## Protected Mode Exceptions

#GP(0) if the current privilege level is larger (has less privilege) than the I/O privilege level and any of the corresponding I/O permission bits in TSS equals 1.

---

## Virtual 8086 Mode Exceptions

#GP(0) fault if any of the corresponding I/O permission bits in TSS equals 1.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Protected Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## INC-Increment by 1

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
FE /0	INC <b>r/m8</b>	X	X	X	X	X	X	Increment r/m byte by 1
FF /0	INC <b>r/m16</b>	X	X	X	X	X	X	Increment r/m word by 1
FF /0	INC <b>r/m32</b>					X	X	Increment r/m dword by 1
40+rw	INC <b>r16</b>	X	X	X	X	X	X	Increment word register by 1
40+rd	INC <b>r32</b>					X	X	Increment dword register by 1

## Description

The INC instruction adds 1 to the operand. It does not change the CF flag. To affect the CF flag, use the ADD instruction with a second operand of 1.

## Operation

DEST ← DEST + 1;

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
*			*	*	*	*	*

The OF, SF, ZF, AF, and PF flags are set according to the result.

## Protected Mode Exceptions

#GP(0) if the operand is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

# Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Related Information

- Description
- Flags Affected
- Operation
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

## INS/INSB/INSW/INSD-Input from Port to String

### Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
6C	INS <code>r/m8,DX</code>	X	X	X	X	X	X	Input byte from port DX into ES: [(E)DI]
6D	INS <code>r/m16,DX</code>	X	X	X	X	X	X	Input word from port DX into ES: [(E)DI]
6D	INS <code>r/m32,DX</code>			X	X	X		Input dword from port DX into ES: [(E)DI]
6C	INSB	X	X	X	X	X	X	Input byte from port DX into ES: [(E)DI]
6D	INSW	X	X	X	X	X	X	Input word from port DX into ES: [(E)DI]
6D	INSD			X	X	X		Input dword from port DX into ES: [(E)DI]

### Description

The INS instruction transfers data from the input port numbered by the DX register to the memory byte or word at ES:dest-index. The memory operand must be addressable from the ES register; no segment override is possible. The destination register is the DI register if the address-size attribute of the instruction is 16-bits, or the EDI register if the address-size attribute is 32-bits.

The INS instruction does not allow the specification of the port number as an immediate value. The port must be addressed through the DX register value. Load the correct value into the DX register before running the INS instruction.

The destination address is determined by the contents of the destination index register. Load the correct index into the destination index register before running the INS instruction.

After the transfer is made, the DI or EDI register advances automatically. If the DF flag is 0 (a CLD instruction was run), the DI or EDI register increments; if the DF flag is 1 (an STD instruction was run), the DI or EDI register decrements. The DI register increments or decrements by 1 if a byte is input, by 2 if a word is input, or by 4 if a doubleword is input.

The INSB, INSW and INSD instructions are synonyms of the byte, word, and doubleword INS instructions. The INS instruction can be preceded by the REP prefix for block input of CX bytes or words. Refer to the REP instruction for details of this operation.

---

## Operation

```
IF AddressSize = 16
THEN use DI for dest-index;
ELSE (* AddressSize = 32 *)
  use EDI for dest-index;
FI;
IF (PE = 1) AND ((VM = 1) OR (CPL>IOPL))
THEN (* Virtual 8086 mode, or protected mode with CPL>IOPL *)
  IF NOT I/O-Permission (SRC, width(SRC))
  THEN #GP(0);
  FI;
FI;
IF byte type of instruction
THEN
  ES:[dest-index] ` [DX]; (* Reads byte at DX from I/O address space *)
  IF DF = 0 THEN IncDec ` 1 ELSE IncDec ` -1; FI;
FI;
IF OperandSize = 16
THEN
  ES:[dest-index] ` [DX]; (* Reads word at DX from I/O address space *)
  IF DF = 0 THEN IncDec ` 2 ELSE IncDec ` -2; FI;
FI;
IF OperandSize = 32
THEN
  ES:[dest-index] ` [DX]; (* Reads dword at DX from I/O address space *)
  IF DF = 0 THEN IncDec ` 4 ELSE IncDec ` -4; FI; FI;
dest-index ` dest-index + IncDec;
```

---

## Protected Mode Exceptions

#GP(0) if the current privilege level is numerically greater than the I/O privilege level and any of the corresponding I/O permission bits in TSS equals 1; #GP(0) if the destination is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the ES, segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

# Virtual 8086 Mode Exceptions

#GP(0) fault if any of the corresponding I/O permission bits in TSS equals 1; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

## INT/INTO-Call to Interrupt Procedure

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
CC	INT 3	X	X	X	X	X	X	Interrupt 3; Trap to debugger
CC	INT 3			X	X	X	X	Interrupt 3; Protected Mode, same privilege
CC	INT 3			X	X	X	X	Interrupt 3; Protected Mode, more privilege
CC	INT 3			X	X	X		Interrupt 3; from V86 mode to PL 0
CC	INT 3			X	X	X	X	Interrupt 3; Protected Mode, via task gate
CC ib	INT imm8	X	X	X	X	X	X	Interrupt numbered by immediate byte
CD ib	INT imm8			X	X	X	X	Interrupt imm8; Protected Mode, same privilege
CD ib	INT imm8			X	X	X	X	Interrupt imm8; Protected Mode, more privilege
CD ib	INT imm8			X	X	X		Interrupt imm8; from V86 mode to PL 0
CD ib	INT imm8			X	X	X	X	Interrupt imm8; Protected Mode, via task gate
CE	INTO	X	X	X	X	X	X	Interrupt 4 if Overflow flag is 1

CE	INTO	X X X X	Interrupt 4 if Overflow flag is 1; Protected Mode, same privilege
CE	INTO	X X X X	Interrupt 4 if Overflow flag is 1; Protected Mode, more privilege
CE	INTO	X X X	Interrupt 4 if Overflow flag is 1; from V86 mode to PL 0
CE	INTO	X X X X	Interrupt 4 if Overflow flag is 1; Protected Mode, via task gate

## Description

The INT *n* instruction generates a call to an interrupt handler. The immediate operand, from 0 to 255, gives the index number into the Interrupt Descriptor Table (IDT) of the interrupt routine to be called. In protected mode, the IDT consists of an array of eight-byte descriptors; the descriptor for the interrupt invoked must indicate an interrupt, trap, or task gate. In real-address mode, the IDT is an array of four byte-long pointers. In protected and real-address modes, the base linear address of the IDT is defined by the contents of the IDTR. The initial value of IDTR is zero upon reset into real-address mode.

When the processor is running in virtual-8086 mode (VM=1), the IOPL determines whether the INT *n* causes a general protection exception (IOPL<3) or runs a protected mode interrupt to privilege level 0. The interrupt gate's DPL must be set to three and the target CPL of the interrupt service routine must be zero to run the protected mode interrupt to privilege level 0.

The INTO conditional software instruction is identical to the INT *n* interrupt instruction except that the interrupt number is implicitly 4, and the interrupt is made only if the overflow flag is set.

The first 32 interrupts are reserved by Intel for system use. Some of these interrupts are used for internally generated exceptions.

The INT *n* instruction generally behaves like a far call except that the flags register is pushed onto the stack before the return address. Interrupt procedures return via the IRET instruction, which pops the flags and return address from the stack.

In Real Address Mode, the INT *n* instruction pushes the flags, the CS register, and the return IP onto the stack, in that order, then jumps to the long pointer indexed by the interrupt number.

## Operation

**Note:** The following operational description applies not only to the above instructions but also to external interrupts and exceptions.

```

IF PE = 0
THEN CALL REAL-ADDRESS-MODE;
ELSE
  CALL PROTECTED-MODE;
  IF task gate
  THEN CALL TASK-GATE;
  ELSE
    CALL TRAP-OR-INT-GATE; (* PE=1, int/trap gate *)
    IF code segment is non-conforming AND DPL < CPL
    THEN
      IF VM=0
      THEN CALL INT-TO-INNER-PRIV; (*PE=1,int/trap gate,DPL<CPL,VM=0*)
      ELSE CALL INT-FROM-V86-MODE; (* PE=1, int/trap gate, DPL<CPL,
        VM=1 *)
    FI;
  ELSE (* PE=1, int/trap gate, DPL = CPL *)
    IF code segment is conforming OR code segment DPL = CPL
    THEN CALL INT-TO-SAME-PRIV;
    ELSE #GP(CS selector + EXT); (* PE=1, int/trap gate, DPL>CPL *)
  FI;
FI;
FI;
FI;
FI;
END;
```

#### REAL-ADDRESS-MODE PROC

```
Push (FLAGS);
IF ` 0; (* Clear interrupt flag *)
TF ` 0; (* Clear trap flag *)
Push(CS);
Push(IP);
(* No error codes are pushed *)
CS ` IDT[Interrupt number * 4].selector;
IP ` IDT[Interrupt number * 4].offset;
(* Start processing in real address mode *)
REAL-ADDRESS-MODE ENDPROC
```

#### PROTECTED-MODE PROC

```
Interrupt vector must be within IDT table limits,
  else #GP(vector number * 8+2+EXT);
Descriptor AR byte must indicate interrupt gate, trap gate, or task gate,
  else #GP(vector number * 8+2+EXT);
IF software interrupt (* i.e. caused by INT n, INT 3, or INTO *)
THEN
  IF gate descriptor DPL<CPL
  THEN #GP(vector number * 8+2+EXT); (* PE=1, DPL<CPL, software interrupt *)
  FI;
FI;
Gate must be present, else #NP(vector number * 8+2+EXT);
PROTECTED-MODE ENDPROC
```

#### TRAP-OR-INT-GATE PROC

```
Examine CS selector and descriptor given in the gate descriptor;
Selector must be non-null, else #GP (EXT);
Selector must be within its descriptor table limits
  ELSE #GP(selector+EXT);
Descriptor AR byte must indicate code segment
  ELSE #GP(selector + EXT);
Segment must be present, else #NP(selector+EXT);
TRAP-OR-INT-GATE ENDPROC
```

#### INT-TO-INNER-PRIV PROC

```
(* PE=1, DPL<CPL and non-conforming, (* PE=1, int/trap gate, DPL<CPL, VM=0 *)
Check selector and descriptor for new stack in current TSS;
Selector must be non-null, else #TS(EXT);
Selector index must be within its descriptor table limits
  ELSE #TS(SS selector+EXT);
Selector's RPL must equal DPL of code segment, else #TS(SS
  selector+EXT);
Stack segment DPL must equal DPL of code segment, else #TS(SS
  selector+EXT);
Descriptor must indicate writable data segment, else #TS(SS
  selector+EXT);
Segment must be present, else #SS (SS selector+EXT);
If 32-bit gate
THEN New stack must have room for 20 bytes else #SS(0)
ELSE New stack must have room for 10 bytes else #SS(0)
FI;
Instruction pointer must be within CS segment boundaries else #GP(0);
Load new SS and eSP value from TSS;
If 32-bit gate
  THEN CS:EIP ` selector:offset from gate;
  ELSE CS:IP ` selector:offset from gate;
FI;
Load CS descriptor into invisible portion of CS register;
Load SS descriptor into invisible portion of SS register;
IF 32-bit gate
THEN
  Push (long pointer to old stack) (* 3 words padded to 4 *);
  Push (EFLAGS);
  Push (long pointer to return location) (* 3 words padded to 4 *);
ELSE
  Push (long pointer to old stack) (* 2 words *);
  Push (FLAGS);
  Push (long pointer to return location) (* 2 words *);
FI;
Set CPL to new code segment DPL;
Set RPL of CS to CPL;
IF interrupt gate THEN IF ` 0 (* interrupt flag to 0 (disable) *); FI;
TF ` 0;
NT ` 0;
```

INT-FROM-INNER-PRIV ENDPROC

INT-FROM-V86-MODE PROC

```
Check selector and descriptor for new stack in current TSS;
Selector must be non-null, else #TS(EXT);
Selector index must be within its descriptor table limits
ELSE #TS(SS selector+EXT);
Selector's RPL must equal DPL of code segment, else #TS(SS
selector+EXT);
Stack segment DPL must equal DPL of code segment, else #TS(SS
selector+EXT);
Descriptor must indicate writable data segment, else #TS(SS
selector+EXT);
Segment must be present, else #SS(SS selector+EXT);
IF 32-bit gate
THEN New stack must have room for 20 bytes else #SS(0)
ELSE New stack must have room for 10 bytes else #SS(0)
FI;
Instruction pointer must be within CS segment boundaries else #GP(0);
IF IOPL < 3
THEN
#GP(0); (*V86 monitor trap: PE=1,int/trap gate, DPL<CPL, VM=1, IOPL<3*)
ELSE (* IOPL=3 *)
IF GATE'S_DPL = 3
THEN
IF TARGET'S_CPL <> 0
THEN #GP(0);
ELSE
TempEFlags ` EFLAGS;
VM ` 0;
TF ` 0;
IF service through Interrupt Gate
THEN IF ` 0;
FI;
TempSS ` SS;
TempESP ` ESP;
SS ` TSS.SS0; (* Change to level 0 stack segment *)
ESP ` TSS.ESP0; (* Change to level 0 stack pointer *)
Push(GS); (* padded to two words *)
Push(FS); (* padded to two words *)
Push(DS); (* padded to two words *)
Push(ES); (* padded to two words *)
GS ` 0; (* segment registers nullified - invalid in
protected mode *)
FS ` 0;
DS ` 0;
ES ` 0;
Push(TempSS); (* Padded to two words *)
Push(TempESP);
Push(TempEFlags);
Push(CS); (* Padded to two words *)
Push(EIP);
CS:EIP ` selector:offset from interrupt gate;
(* Starts processing of new routine in Protected Mode *)
FI;
ELSE (* GATE'S_DPL <> 3 *)
#GP(0);
FI;
FI;
```

INT-FROM-V86-MODE ENDPROC

INT-TO-SAME-PRIV PROC

```
(* PE=1, DPL=CPL or conforming segment *)
IF 32-bit gate
THEN Current stack limits must allow pushing 10 bytes, else #SS(0);
ELSE Current stack limits must allow pushing 6 bytes, else #SS(0);
FI;
```

```
IF interrupt was caused by exception with error code
THEN Stack limits must allow push to two more bytes;
ELSE #SS(0);
FI;
Instruction pointer must be in CS limit, else #GP(0);
IF 32-bit gate
THEN
Push (EFLAGS);
```

```

    Push (long pointer to return location); (* 3 words padded to 4 *)
    CS:EIP ` selector:offset from gate;
ELSE (* 16-bit gate *)
    Push (FLAGS);
    Push (long pointer to return location); (* 2 words *)
    CS:IP ` selector:offset from gate;
FI;
Load CS descriptor into invisible portion of CS register;
Set the RPL field of CS to CPL;
Push (error code); (* if any *)
IF interrupt gate THEN IF ` 0; FI;
TF ` 0;
NT ` 0;
INT-TO-SAME-PRIV ENDPROC

```

```

TASK-GATE PROC (* PE=1, task gate *)
    Examine selector to TSS, given in task gate descriptor;
    Must specify global in the local/global bit, else #TS(TSS selector);
    Index must be within GDT limits, else #TS(TSS selector);
    AR byte must specify available TSS (bottom bits 00001);
    else #TS(TSS selector);
    TSS must be present, else #NP(TSS selector);
    SWITCH-TASKS with nesting to TSS;
    IF interrupt was caused by fault with error code
    THEN
        Stack limits must allow push of two more bytes, else #SS(0);
        Push error code onto stack;
    FI;
    Instruction pointer must be in CS limit, else #GP(0);
TASK-GATE ENDPROC

```

## Decision Table

The following decision table indicates which action in the lower portion of the table is taken given the conditions in the upper portion of the table. Each Y in the lower section of the decision table represents a procedure defined above in the Operation section for this instruction (except #GP(0)) and the number following the Y indicates the order in which the procedure is run.

PE	0	1	1	1	1	1	1	1
VM	-	-	-	-	-	0	1	1
IOPL	-	-	-	-	-	-	<3	=3
DPL/CPL RELATIONSHIP	-	DPL < CPL	-	DPL > CPL	DPL = CPL or C	DPL < CPL & NC	-	-
INTERRUPT TYPE	-	S/W	-	-	-	-	-	-
GATE TYPE	-	-	Task	Trap or Int	Trap or Int	Trap or Int	Trap or Int	Trap or Int
REAL-ADDRESS-MODE	Y							
PROTECTED-MODE		Y1	Y1	Y1	Y1	Y1	Y1	Y1
TRAP-OR-INT-GATE				Y2	Y2	Y2	Y2	Y2
INT-TO-INNER-PRIV						Y3		
INT-TO-SAME-PRIV					Y3			
INT-FROM-V86-MODE								Y3
TASK-GATE			Y2					
#GP		Y2		Y3			Y3	

**Notes:**

-	Don't Care
Yx	Yes, Action Taken, x = the order of processing
Blank	Action Not Taken

-----

## Flags Affected

OF DF IF SF ZF AF PF CF

0

None

-----

## Protected Mode Exceptions

#GP, #NP, #SS, and #TS as indicated under "Operation" above.

-----

## Real Address Mode Exceptions

None; if the SP or ESP register is 1, 3, or 5 before running the INT or INTO instruction, the processor will shut down due to insufficient stack space.

-----

## Virtual 8086 Mode Exceptions

#GP(0) fault if IOPL is less than 3, for the INT *n* instruction only, to permit emulation; Interrupt 3 (0CCH) generates a breakpoint exception; the INTO instruction generates an overflow exception if the OF flag is set.

-----

## Related Information

[Decision Table](#)

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

# INVD-Invalidate Cache

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F 08	INVD					X	X	Invalidate entire cache

---

## Description

The internal cache is invalidated, and a special-function bus cycle is issued which indicates that external caches should also be invalidated. Data held in write-back external caches is not instructed to be written back.

---

## Operation

INVALIDATE INTERNAL CACHE  
SIGNAL EXTERNAL CACHE TO INVALIDATE

---

## Protected Mode Exceptions

The INVD instruction is a privileged instruction; #GP(0) if the current privilege level is not 0.

---

## Virtual 8086 Mode Exceptions

#GP(0); the INVD instruction is a privileged instruction.

---

## Notes

INVD should be used with care. It does not write back modified cache lines; therefore, it can cause the data cache to become inconsistent with other memories in the system. Unless there is a specific requirement or benefit to invalidate a cache without writing back the modified

lines (i.e., testing or fault recovery where cache coherency with main memory is not a concern), software should use the WBINVD instruction.

This instruction is implementation-dependent; its function may be implemented differently on future Intel processors.

This instruction does not wait for the external cache to complete its invalidation before the processor proceeds. It is the responsibility of hardware to respond to the external cache invalidation indication.

This instruction is not supported on Intel386 processors. See the Intel documentation for CPUID detection at runtime. See the WBINVD description to write back dirty data to memory.

See the Intel documentation for information on disabling the cache.

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Protected Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

## INVLPG-Invalidate TLB Entry

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F 01 /7	INVLPG <i>m</i>					<i>X</i>	<i>X</i>	Invalidate TLB entry

## Description

The INVLPG instruction is used to ensure there are no invalid entries in the TLB, the cache used for page table entries. If the TLB contains a valid entry, which maps the address of the memory operand, all of the relevant TLB entries are marked invalid.

## Operation

INVALIDATE RELEVANT TLB ENTRY(S)

---

## Protected Mode Exceptions

The INVLPG instruction is a privileged instruction; #GP(0) if the current privilege level is not 0. An invalid-opcode exception is generated when used with a register operand.

---

## Virtual 8086 Mode Exceptions

An invalid-opcode exception is generated when used with a register operand. #GP(0); the INVLPG instruction is a privileged instruction.

---

## Notes

This instruction is not supported on Intel386 processors. See the Intel documentation for information on detecting the processor type at runtime.

See the Intel documentation for information on disabling the cache.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Protected Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## IRET/IRETD-Interrupt Return

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
CF	IRET	X	X	X	X	X	X	Interrupt return (far return and pop flags)
CF	IRET		X	X	X	X	X	Interrupt return to lesser privilege
CF	IRET		X	X	X	X	X	Interrupt return, different task (NT=1)
CF	IRET		X	X	X	X		Interrupt return from Real or V86 mode
CF	IRETD		X	X	X	X		Interrupt return (far return and pop flags)
CF	IRETD		X	X	X	X		Interrupt return to lesser privilege
CF	IRETD		X	X	X	X		Interrupt return, different task (NT=1)

-----

## Description

In Real Address Mode, the IRET instruction pops the instruction pointer, the CS register, and the flags register from the stack and resumes the interrupted routine.

In Protected Mode, the action of the IRET instruction depends on the setting of the nested task flag (NT) bit in the flag register. When the new flag image is popped from the stack, the IOPL bits in the flag register are changed only when CPL equals 0.

If the NT flag is cleared, the IRET instruction returns from an interrupt procedure without a task switch. The code returned to must be equally or less privileged than the interrupt routine (as indicated by the RPL bits of the CS selector popped from the stack). If the destination code is less privileged, the IRET instruction also pops the stack pointer and SS from the stack.

If the NT flag is set, the IRET instruction reverses the operation of a CALL or INT that caused a task switch. The updated state of the task running the IRET instruction is saved in its task state segment. If the task is reentered later, the code that follows the IRET instruction is run.

-----

## Operation

```

IF PE = 0
THEN GOTO REAL_ADDRESS_MODE;;
ELSE GOTO PROTECTED_MODE;
FI;

REAL_ADDRESS_MODE;
IF OperandSize = 32 (* Instruction = IRETD *)
THEN EIP ` Pop ();
ELSE (* Instruction = IRET *)
IP ` Pop();
FI;
CS ` Pop ();
IF OperandSize = 32 (* Instruction = IRETD *)
THEN Pop(); EFLAGS ` Pop();
ELSE (* Instruction = IRET *)
FLAGS ` Pop();
FI;
END;

PROTECTED_MODE:
IF VM = 1 (* Virtual mode:PE=1, VM=1 *)
THEN GOTO STACK_RETURN_FROM_V86; (* PE=1, VM=1 *)
ELSE

```

```

IF NT=1
THEN GOTO TASK_RETURN; (* PE=1, VM=1, NT=1 *)
ELSE
  IF VM=1 in flags image on stack
  THEN GOTO STACK_RETURN_TO_V86; (* PE=1, VM=1 in flags
    image *)
  ELSE GOTO STACK_RETURN; (* PE=1, VM=0 in flags image *)
FI;
FI;
FI;

STACK_RETURN_FROM_V86:
IF IOPL=3 (* Virtual mode: PE=1, VM=1, IOPL=3 *)
THEN
  IF OperandSize = 16
  IP ` Pop();(* 16-bit pops *)
  CS ` Pop();
  FLAGS ` Pop();
  ELSE (* OperandSize = 32 *)
  EIP ` Pop(); (* 32-bit pops *)
  CS ` Pop();
  EFLAGS ` Pop(); (*VM,IOPL,VIP,and VIF EFLAG bits are not modified by IRETD*)
  FI;
  ELSE #GP(0); (* trap to virtual-8086 monitor: PE=1, VM=1, IOPL<3 *)
  FI;
END;

```

```

STACK_RETURN_TO_V86: (* Interrupted procedure was in V86 mode:
PE=1, VM=1 in flags image *)
IF top 36 bytes of stack not within limits
THEN #SS(0);
FI;
IF instruction pointer not within code segment limit THEN #GP(0);
FI;
EFLAGS ` SS:[ESP + 8]; (* Sets VM in interrupted routine *)
EIP ` Pop();
CS ` Pop(); (* CS behaves as in 8086, due to VM=1 *)
throwaway ` Pop(); (* pop away EFLAGS already read *)
TempESP ` Pop();
TempSS ` Pop();
ES ` Pop(); (* pop 2 words; throw away high-order word *)
DS ` Pop(); (* pop 2 words; throw away high-order word *)
FS ` Pop(); (* pop 2 words; throw away high-order word *)
GS ` Pop(); (* pop 2 words; throw away high-order word *)
SS:ESP ` TempSS:TempESP;
(* Resume processing in Virtual 8086 mode *)
END;

```

```

TASK-RETURN: (* PE=1, VM=1, NT=1 *)
Examine Back Link Selector in TSS addressed by the current task
  register:
  Must specify global in the local/global bit, else #TS(new TSS selector);
  Index must be within GDT limits, else #TS(new TSS selector);
  AR byte must specify TSS, else #TS(new TSS selector);
  New TSS must be busy, else #TS(new TSS selector);
  TSS must be present, else #NP(new TSS selector);
SWITCH-TASKS without nesting to TSS specified by back link selector;
Mark the task just abandoned as NOT BUSY;
Instruction pointer must be within code segment limit ELSE #GP(0);
END;

```

```

STACK-RETURN: (* PE=1, VM=0 in flags image *)
IF OperandSize=32
THEN Third word on stack must be within stack limits, else #SS(0);
ELSE Second word on stack must be within stack limits, else #SS(0);
FI;
Return CS selector RPL must be CPL, else #GP(Return selector);
IF return selector RPL = CPL
THEN GOTO RETURN-SAME-LEVEL;
ELSE GOTO RETURN-OTHER-LEVEL;
FI;

```

```

RETURN-SAME-LEVEL: (* PE=1, VM=0 in flags image, RPL=CPL *)
IF OperandSize=32
THEN
  Top 12 bytes on stack must be within limits, else #SS(0);

```

```

    Return CS selector (at eSP+4) must be non-null, else #GP(0);
ELSE
    Top 6 bytes on stack must be within limits, else #SS(0);
    Return CS selector (at eSP+2) must be non-null, else #GP(0);
FI;
Selector index must be within its descriptor table limits, else #GP
(Return selector);
AR byte must indicate code segment, else #GP(Return selector);
IF non-conforming
THEN code segment DPL must = CPL;
ELSE #GP(Return selector); (* PE=1, VM=0 in flags image,
RPL=CPL,non-conforming,DPL<> CPL *)
FI;
IF conforming
THEN IF DPL>CPL
    #GP(Return selector); (* PE=1, VM=0 in flags image,
    RPL=CPL,conforming,DPL>CPL *)
Segment must be present, else #NP(Return selector);
Instruction pointer must be within code segment boundaries, else #GP(0);
FI;
IF OperandSize=32 put comments here
THEN
    Load CS:EIP from stack;
    Load CS-register with new code segment descriptor;
    Load EFLAGS with third doubleword from stack;
    Increment eSP by 12;
ELSE
    Load CS-register with new code segment descriptor;
    Load FLAGS with third word on stack;
    Increment eSP by 6;
FI;
END;

```

RETURN-OUTER-LEVEL:

```

IF OperandSize=32
THEN Top 20 bytes on stack must be within limits, else #SS(0);
ELSE Top 10 bytes on stack must be within limits, else #SS(0);
FI;
Examine return CS selector and associated descriptor:
    Selector must be non-null, ELSE #GP(0);
    Selector index must be within its descriptor table limits;
    ELSE #GP(Return selector);
    AR byte must indicate code segment, else #GP(Return selector);
IF non-conforming
THEN code segment DPL must = CS selector RPL;
ELSE #GP(Return selector);
FI;
IF conforming
THEN code segment DPL must be > CPL;
ELSE #GP(Return selector);
FI;
Segment must be present, else #NP(Return selector);
Examine return SS selector and associated descriptor:
    Selector must be non-null, else #GP(0);
    Selector index must be within its descriptor table limits
    ELSE #GP(SS selector);
    Selector RPL must equal the RPL of the return CS selector
    ELSE #GP(SS selector);
    AR byte must indicate a writable data segment, else #GP(SS selector);
    Stack segment DPL must equal the RPL of the return CS selector
    ELSE #GP(SS selector);
    SS must be present, else #NP(SS selector);
Instruction pointer must be within code segment limit ELSE #GP(0);
IF OperandSize=32
THEN
    Load CS:EIP from stack;
    Load EFLAGS with values at (eSP+8);
ELSE
    Load CS:IP from stack;
    Load FLAGS with values at (eSP+4);
FI;
Load SS:eSP from stack;
Set CPL to the RPL of the return CS selector;
Load the CS register with the CS descriptor;
Load the SS register with the SS descriptor;
FOR each of ES, FS, GS, and DS

```

DO;  
IF the current value of the register is not valid for the outer level;  
THEN zero the register and clear the valid flag;  
FI;  
To be valid, the register setting must satisfy the following properties:  
Selector index must be within descriptor table limits;  
AR byte must indicate data or readable code segment;  
IF segment is data or non-conforming code,  
THEN DPL must be > CPL, or DPL must be < RPL;  
OD;  
END:

-----

## Flags Affected

OF DF IF SF ZF AF PF CF  
\* \* \* \* \*

All flags are affected; the flags register is popped from stack.

-----

## Protected Mode Exceptions

#GP, #NP, or #SS, as indicated under "Operation" above; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand being popped lies beyond address 0FFFFH.

-----

## Virtual 8086 Mode Exceptions

#GP(0) fault occurs if the I/O privilege level is less than 3, to permit emulation; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

# Jcc-Jump if Condition is Met

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
77 cb	JA <a href="#">rel8</a>	X	X	X	X	X	X	Jump short if above (CF=0 and ZF=0)
73 cb	JAE <a href="#">rel8</a>	X	X	X	X	X	X	Jump short if above or equal (CF=0)
72 cb	JB <a href="#">rel8</a>	X	X	X	X	X	X	Jump short if below (CF=1)
76 cb	JBE <a href="#">rel8</a>	X	X	X	X	X	X	Jump short if below or equal (CF=1 or ZF=1)
72 cb	JC <a href="#">rel8</a>	X	X	X	X	X	X	Jump short if carry (CF=1)
E3 cb	JCXZ <a href="#">rel8</a>	X	X	X	X	X	X	Jump short if CX register is 0
E3 cb	JECXZ <a href="#">rel8</a>			X	X	X		Jump short if ECX register is 0
74 cb	JE <a href="#">rel8</a>	X	X	X	X	X	X	Jump short if equal (ZF=1)
74 cb	JG <a href="#">rel8</a>	X	X	X	X	X	X	Jump short if greater (ZF=0 and SF=OF)
7D cb	JGE <a href="#">rel8</a>	X	X	X	X	X	X	Jump short if greater or equal (SF=OF)
7C cb	JL <a href="#">rel8</a>	X	X	X	X	X	X	Jump short if less (SF<>OF)
7E cb	JLE <a href="#">rel8</a>	X	X	X	X	X	X	Jump short if less or equal (ZF=1 and SF<>OF)
76 cb	JNA <a href="#">rel8</a>	X	X	X	X	X	X	Jump short if not above (CF=1 or ZF=1)
72 cb	JNAE <a href="#">rel8</a>	X	X	X	X	X	X	Jump short if not above or equal (CF=1)
73 cb	JNB <a href="#">rel8</a>	X	X	X	X	X	X	Jump short if not below (CF=0)
77 cb	JNBE <a href="#">rel8</a>	X	X	X	X	X	X	Jump short if not below or equal (CF=0 and ZF=0)
73 cb	JNC <a href="#">rel8</a>	X	X	X	X	X	X	Jump short if not carry (CF=0)
75 cb	JNE <a href="#">rel8</a>	X	X	X	X	X	X	Jump short if not equal (ZF=0)
7E cb	JNG <a href="#">rel8</a>	X	X	X	X	X	X	Jump short if not greater (ZF=1 or SF<>OF)
7C cb	JNGE <a href="#">rel8</a>	X	X	X	X	X	X	Jump short if not greater or equal (SF<>OF)
7D cb	JNL <a href="#">rel8</a>	X	X	X	X	X	X	Jump short if not less (SF=OF)
7F cb	JNLE <a href="#">rel8</a>	X	X	X	X	X	X	Jump short if not less or equal (ZF=0 and SF=OF)

71 cb	JNO <a href="#">rel8</a>	X X X X X X	Jump short if not overflow (OF=0)
7B cb	JNP <a href="#">rel8</a>	X X X X X X	Jump short if not parity (PF=0)
79 cb	JNS <a href="#">rel8</a>	X X X X X X	Jump short if not sign (SF=0)
75 cb	JNZ <a href="#">rel8</a>	X X X X X X	Jump short if not zero (ZF=0)
70 cb	JO <a href="#">rel8</a>	X X X X X X	Jump short if overflow (OF=1)
7A cb	JP <a href="#">rel8</a>	X X X X X X	Jump short if parity (PF=1)
7A cb	JPE <a href="#">rel8</a>	X X X X X X	Jump short if parity even (PF=1)
7B cb	JPO <a href="#">rel8</a>	X X X X X X	Jump short if parity odd (PF=0)
78 cb	JS <a href="#">rel8</a>	X X X X X X	Jump short if sign (SF=1)
74 cb	JZ <a href="#">rel8</a>	X X X X X X	Jump short if zero (ZF=1)
0F 87 cw/cd	JA <a href="#">rel16/rel32</a>	X X X	Jump near if above (CF=0 and ZF=0)
0F 83 cw/cd	JAE <a href="#">rel16/rel32</a>	X X X	Jump near if above or equal (CF=0)
0F 82 cw/cd	JB <a href="#">rel16/rel32</a>	X X X	Jump near if below (CF=1)
0F 86 cw/cd	JBE <a href="#">rel16/rel32</a>	X X X	Jump near if below or equal (CF=1 or ZF=1)
0F 82 cw/cd	JC <a href="#">rel16/rel32</a>	X X X	Jump near if carry (CF=1)
0F 84 cw/cd	JE <a href="#">rel16/rel32</a>	X X X	Jump near if equal (ZF=1)
0F 8F cw/cd	JG <a href="#">rel16/rel32</a>	X X X	Jump near if greater (ZF=0 and SF=OF)
0F 8D cw/cd	JGE <a href="#">rel16/rel32</a>	X X X	Jump near if greater or equal (SF=OF)
0F 8C cw/cd	JL <a href="#">rel16/rel32</a>	X X X	Jump near if less (SF<>OF)
0F 8E cw/cd	JLE <a href="#">rel16/rel32</a>	X X X	Jump near if less or equal (ZF=1 and SF<>OF)
0F 86 cw/cd	JNA <a href="#">rel16/rel32</a>	X X X	Jump near if not above (CF=1 or ZF=1)
0F 82 cw/cd	JNAE <a href="#">rel16/rel32</a>	X X X	Jump near if not above or equal (CF=1)
0F 83 cw/cd	JNB <a href="#">rel16/rel32</a>	X X X	Jump near if not below (CF=0)
0F 87 cw/cd	JNBE <a href="#">rel16/rel32</a>	X X X	Jump near if not below or equal (CF=0 and ZF=0)
0F 83 cw/cd	JNC <a href="#">rel16/rel32</a>	X X X	Jump near if not carry (CF=0)
0F 85 cw/cd	JNE <a href="#">rel16/rel32</a>	X X X	Jump near if not equal (ZF=0)
0F 8E cw/cd	JNG <a href="#">rel16/rel32</a>	X X X	Jump near if not greater (ZF=1 or SF<>OF)
0F 8C cw/cd	JNGE <a href="#">rel16/rel32</a>	X X X	Jump near if not greater or equal (SF<>OF)
0F 8D cw/cd	JNL <a href="#">rel16/rel32</a>	X X X	Jump near if not less (SF=OF)
0F 8F cw/cd	JNLE <a href="#">rel16/rel32</a>	X X X	Jump near if not less or equal (ZF=0 and SF=OF)
0F 81 cw/cd	JNO <a href="#">rel16/rel32</a>	X X X	Jump near if not overflow (OF=0)

0F 8B cw/cd	JNP <a href="#">rel16/rel32</a>	X X X Jump near if not parity (PF=0)
0F 89 cw/cd	JNS <a href="#">rel16/rel32</a>	X X X Jump near if not sign (SF=0)
0F 85 cw/cd	JNZ <a href="#">rel16/rel32</a>	X X X Jump near if not zero (ZF=0)
0F 80 cw/cd	JO <a href="#">rel16/rel32</a>	X X X Jump near if overflow (OF=1)
0F 8A cw/cd	JP <a href="#">rel16/rel32</a>	X X X Jump near if parity (PF=1)
0F 8A cw/cd	JPE <a href="#">rel16/rel32</a>	X X X Jump near if parity even (PF=1)
0F 8B cw/cd	JPO <a href="#">rel16/rel32</a>	X X X Jump near if parity odd (PF=0)
0F 88 cw/cd	JS <a href="#">rel16/rel32</a>	X X X Jump near if sign (SF=1)
0F 84 cw/cd	JZ <a href="#">rel16/rel32</a>	X X X Jump near if zero (ZF=1)

-----

## Description

Conditional jumps (except the JCXZ instruction) test the flags that have been set by a previous instruction. The conditions for each mnemonic are given in parentheses after each description above. The terms "less" and "greater" are used for comparisons of signed integers; "above" and "below" are used for unsigned integers.

If the given condition is true, a jump is made to the location provided as the operand. Instruction coding is most efficient when the target for the conditional jump is in the current code segment and within -128 to +127 bytes of the next instruction's first byte. The jump can also target -32768 thru +32767 (segment size attribute 16) or -2<sup>31</sup> thru +2<sup>31</sup> - 1 (segment size attribute 32) relative to the next instruction's first byte. When the target for the conditional jump is in a different segment, use the opposite case of the jump instruction (i.e., the JE and JNE instructions), and then access the target with an unconditional far jump to the other segment. For example, you cannot code-

```
JZ FARLABEL
```

You must instead code-

```
JNZ BEYOND
JMP FARLABEL
```

BEYOND:

Because there can be several ways to interpret a particular state of the flags, ASM386 provides more than one mnemonic for most of the conditional jump opcodes. For example, if you compared two characters in AX and want to jump if they are equal, use the JE instructions; or, if you ANDed the AX register with a bit field mask and only want to jump if the result is 0, use the JZ instruction, a synonym for the JE instruction.

The JCXZ instruction differs from other conditional jumps because it tests the contents of the CX or ECX register for 0, not the flags. The JCXZ instruction is useful at the beginning of a conditional loop that terminates with a conditional loop instruction (such as LOOPNE TARGET LABEL). The JCXZ instruction prevents entering the loop with the CX or ECX register equal to zero, which would cause the loop to run 64K or 2<sup>32</sup> times instead of zero times.

-----

## Operation

IF condition

THEN

```
EIP ← EIP + SignExtend(rel8/16/32);
```

```
IF OperandSize = 16
```

```
THEN EIP ← EIP AND 0000FFFFH;
```

```
FI;
```

FI;

# Protected Mode Exceptions

#GP(0) if the offset jumped to is beyond the limits of the code segment.

## Notes

The JCXZ instruction takes longer to run than a two-instruction sequence, which compares the count register to zero and jumps if the count is zero.

All branches are converted into 16-byte code fetches regardless of jump address or cacheability.

## Related Information

- Description
- Flags Affected
- Notes
- Operation
- Protected Mode Exceptions
- Protected Mode Exceptions
- Virtual 8086 Mode Exceptions

## JMP-Jump

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
EB cb	JMP <a href="#">rel8</a>	X	X	X	X	X	X	Jump short, displacement relative to next instruction
E9 cw	JMP <a href="#">rel16</a>	X	X	X	X	X	X	Jump near, displacement relative to next instruction
FF /4	JMP <a href="#">r/m16</a>	X	X	X	X	X	X	Jump near indirect
EA cd	JMP <a href="#">ptr16:16</a>	X	X	X	X	X	X	Jump intersegment, 4-byte immediate address

EA cd	JMP <i>ptr16:16</i>	X X X X	Jump to call gate, same privilege
EA cd	JMP <i>ptr16:16</i>	X X X X	Jump via task state segment
EA cd	JMP <i>ptr16:16</i>	X X X X	Jump via task gate
FF /5	JMP <i>m16:16</i>	X X X X X X	Jump intersegment, dword address at r/m word
FF /5	JMP <i>m16:16</i>	X X X X	Jump to call gate, same privilege
FF /5	JMP <i>m16:16</i>	X X X X	Jump via task state segment
FF /5	JMP <i>m16:16</i>	X X X X	Jump via task gate
E9 cd	JMP <i>rel32</i>	X X X	Jump near, displacement relative to next instruction
FF /4	JMP <i>r/m32</i>	X X X	Jump near indirect
EA cp	JMP <i>ptr16:32</i>	X X X	Jump intersegment, 6-byte immediate address
EA cp	JMP <i>ptr16:32</i>	X X X	Jump to call gate, same privilege
EA cp	JMP <i>ptr16:32</i>	X X X	Jump via task state segment
EA cp	JMP <i>ptr16:32</i>	X X X	Jump via task gate
FF /5	JMP <i>m16:32</i>	X X X	Jump intersegment, fword address at r/m dword
FF /5	JMP <i>m16:32</i>	X X X	Jump to call gate, same privilege
FF /5	JMP <i>m16:32</i>	X X X	Jump via task state segment
FF /5	JMP <i>m16:32</i>	X X X	Jump via task gate

-----

## Description

The JMP instruction transfers control to a different point in the instruction stream without recording return information.

The action of the various forms of the instruction are shown below.

Jumps with destinations of type *r/m16*, *r/m32*, *rel16*, , and *rel32* are near jumps and do not involve changing the segment register value.

The JMP *rel16* and JMP *rel32* forms of the instruction add an offset to the address of the instruction following the JMP to determine the destination. The *rel16* form is used when the instruction's operand-size attribute is 16-bits (segment size attribute 16 only); *rel32* is used when the operand-size attribute is 32-bits (segment size attribute 32 only). The result is stored in the 32-bit EIP register. With *rel16*, the upper 16-bits of the EIP register are cleared, which results in an offset whose value does not exceed 16-bits.

The JMP *r/m16* and JMP *r/m32* forms specify a register or memory location from which the absolute offset from the procedure is fetched. The offset fetched from *r/m* is 32-bits for an operand-size attribute of 32-bits (*r/m32*), or 16-bits for an operand-size attribute of 16-bits (*r/m16*).

The JMP *ptr16:16* and *ptr16:32* forms of the instruction use a four-byte or six-byte operand as a long pointer to the destination. The JMP *m16:16* and *m16:32* forms fetch the long pointer from the memory location specified (indirection). In Real Address Mode or Virtual 8086 Mode, the long pointer provides 16-bits for the CS register and 16 or 32-bits for the EIP register (depending on the operand-size attribute). In Protected Mode, both long pointer forms consult the Access Rights (AR) byte in the descriptor indexed by the selector part of the long pointer. Depending on the value of the AR byte, the jump will perform one of the following types of control transfers:

- A jump to a code segment at the same privilege level
- A task switch

For more information on protected mode control transfers, refer to the Intel documentation.

-----

## Operation

```

IF instruction = relative JMP (* i.e. operand is rel/8, rel/16, or rel/32 *)
THEN
  EIP ` EIP + rel/18/16/32;
  IF OperandSize = 16
    THEN EIP ` EIP AND 0000FFFFH;
  FI;
FI;

```

```

IF instruction = near indirect JMP
(* i.e. operand is r/m16 or r/m32 *)
THEN
  IF OperandSize = 16
    THEN
      EIP ` [r/m16] AND 0000FFFFH;
    ELSE (* OperandSize = 32 *)
      EIP ` [r/m32];
    FI;
  FI;
FI;

```

```

IF (PE = 0 OR (PE = 1 AND VM = 1)) (* real mode or V86 mode *)
  AND instruction = far JMP
  (* i.e., operand type is m16:16, m16:32, ptr16:16, ptr16:32 *)
THEN GOTO REAL-OR-V86-MODE;
IF operand type = m16:16 or m16:32
  THEN (* indirect *)
    IF OperandSize = 16
      THEN
        CS:IP ` [m16:16];
        EIP ` EIP AND 0000FFFFH; (* clear upper 16 bits *)
      ELSE (* OperandSize = 32 *)
        CS:EIP ` [m16:32];
      FI;
    FI;
  IF operand type = ptr16:16 or ptr16:32
    THEN
      IF OperandSize = 16
        THEN
          CS:IP ` ptr16:16;
          EIP ` EIP AND 0000FFFFH; (* clear upper 16 bits *)
        ELSE (* OperandSize = 32 *)
          CS:EIP ` ptr16:32;
        FI;
      FI;
    FI;
  FI;

```

```

IF (PE = 1 AND VM = 0) (* Protected mode, not V86 mode *)
  AND instruction = far JMP
THEN
  IF operand type = m16:16 or m16:32
    THEN (* indirect *)
      check access of EA dword;
      #GP(0) or #SS(0) IF limit violation;
    FI;
    Destination selector is not null ELSE #GP(0)
    Destination selector index is within its descriptor table limits ELSE #GP(selector)
    Depending on AR byte of destination descriptor;
    GOTO CONFORMING-CODE-SEGMENT;
    GOTO NONCONFORMING-CODE-SEGMENT;
    GOTO CALL-GATE;
    GOTO TASK-GATE;
    GOTO TASK-STATE-SEGMENT;
    ELSE #GP(selector); (* illegal AR byte in descriptor *)
  FI;
FI;

```

```

CONFORMING-CODE-SEGMENT:
  Descriptor DPL must be %4 CPL ELSE #GP(selector);
  Segment must be present ELSE #NP(selector);
  Instruction pointer must be within code-segment limit ELSE #GP(0);
  IF OperandSize = 32
    THEN Load CS:EIP from destination pointer;
  ELSE Load CS:IP from destination pointer;
  FI;
  Load CS register with new segment descriptor;

```

#### NONCONFORMING-CODE-SEGMENT:

RPL of destination selector must be CPL ELSE #GP(selector);  
Descriptor DPL must be = CPL ELSE #GP(selector);  
Segment must be present ELSE #NP(selector);  
Instruction pointer must be within code-segment limit ELSE #GP(0);  
IF OperandSize = 32  
THEN Load CS:EIP from destination pointer;  
ELSE Load CS:IP from destination pointer;  
FI;  
Load CS register with new segment descriptor;  
Set RPL field of CS register to CPL;

#### CALL-GATE:

Descriptor DPL must be CPL ELSE #GP(gate selector);  
Descriptor DPL must be gate selector RPL ELSE #GP(gate selector);  
Gate must be present ELSE #NP(gate selector);  
Examine selector to code segment given in call gate descriptor:  
Selector must not be null ELSE #GP(0);  
Selector must be within its descriptor table limits ELSE  
#GP(CS selector);  
Descriptor AR byte must indicate code segment  
ELSE #GP(CS selector);  
IF non-conforming  
THEN code-segment descriptor DPL must = CPL  
ELSE #GP(CS selector);  
FI;  
IF conforming  
THEN code-segment descriptor DPL must be  $\frac{3}{4}$  CPL;  
ELSE #GP(CS selector);  
Code segment must be present ELSE #NP(CS selector);  
Instruction pointer must be within code-segment limit ELSE #GP(0);  
IF OperandSize = 32  
THEN Load CS:EIP from call gate;  
ELSE Load CS:IP from call gate;  
FI;  
Load CS register with new code-segment descriptor;  
Set RPL of CS to CPL

#### TASK-GATE:

Gate descriptor DPL must be CPL ELSE #GP(gate selector);  
Gate descriptor DPL must be gate selector RPL ELSE #GP(gate selector);  
Task Gate must be present ELSE #NP(gate selector);  
Examine selector to TSS, given in Task Gate descriptor:  
Must specify global in the local/global bit ELSE #GP(TSS selector);  
Index must be within GDT limits ELSE #GP(TSS selector);  
Descriptor AR byte must specify available TSS (bottom bits 00001);  
ELSE #GP(TSS selector);  
Task State Segment must be present ELSE #NP(TSS selector);  
SWITCH-TASKS (without nesting) to TSS;  
Instruction pointer must be within code-segment limit ELSE #GP(0);

#### TASK-STATE-SEGMENT:

TSS DPL must be CPL ELSE #GP(TSS selector);  
TSS DPL must be TSS selector RPL ELSE #GP(TSS selector);  
Descriptor AR byte must specify available TSS (bottom bits 00001)  
ELSE #GP(TSS selector);  
Task State Segment must be present ELSE #NP(TSS selector);  
SWITCH-TASKS (without nesting) to TSS;  
Instruction pointer must be within code-segment limit ELSE #GP(0);

-----

## Flags Affected

OF DF IF SF ZF AF PF CF

All if a task switch takes place; none if no task switch occurs.

---

## Protected Mode Exceptions

Far jumps: #GP, #NP, #SS, and #TS, as indicated in the list above.

Near direct jumps: #GP(0) if procedure location is beyond the code segment limits; #AC for unaligned memory reference if the current privilege level is 3.

Near indirect jumps: #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #GP if the indirect offset obtained is beyond the code segment limits; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would be outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as under Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

All branches are converted into 16-byte code fetches regardless of jump address or cacheability.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## LAHF-Load Flags into AH Register

---

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
9F	LAHF	X	X	X	X	X	X	AH ` flags (SF, ZF, xx, AF, xx, PF, xx, CF)

---

## Description

The LAHF instruction transfers the low byte of the flags word to the AH register. The bits, from MSB to LSB, are sign, zero, indeterminate, auxiliary, carry, indeterminate, parity, indeterminate, and carry.

---

## Operation

AH ` SF:ZF:xx:AF:xx:PF:xx:CF;

---

## Related Information

- [Description](#)
- [Flags Affected](#)
- [Operation](#)
- [Protected Mode Exceptions](#)
- [Protected Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

---

## LAR-Load Access Rights Byte

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F 02 /r	LAR <b>r16,r/m16</b>			<b>X</b>	<b>X</b>	<b>X</b>	<b>X</b>	r16 ` r/m16 masked by FF00
0F 02 /r	LAR <b>r32,r/m32</b>			<b>X</b>	<b>X</b>	<b>X</b>		r32 ` r/m32 masked by 00F?FF00

-----

## Description

The LAR instruction stores a marked form of the second doubleword of the descriptor for the source selector if the selector is visible at the current privilege level (modified by the selector's RPL) and is a valid descriptor type within the descriptor limits. The destination register is loaded with the high-order doubleword of the descriptor masked by 00FxFF00, and the ZF flag is set. The x indicates that the four bits corresponding to the upper four bits of the limit are undefined in the value loaded by the LAR instruction. If the selector is invisible or of the wrong type, the ZF flag is cleared.

If the 32-bit operand size is specified, the entire 32-bit value is loaded into the 32-bit destination register. If the 16-bit operand size is specified, the lower 16-bits of this value are stored in the 16-bit destination register.

All code and data segment descriptors are valid for the LAR instruction.

The valid special segment and gate descriptor types for the LAR instruction are given in the following table:

TYPE	NAME	VALID/INVALID
0	Invalid	Invalid
1	Available 16-bit TSS	Valid
2	LDT	Valid
3	Busy 16-bit TSS	Valid
4	16-bit call gate	Valid
5	16-bit/32-bit task gate	Valid
6	16-bit trap gate	Invalid
7	16-bit interrupt gate	Invalid
8	Invalid	Invalid
9	Available 32-bit TSS	Valid
A	Invalid	Invalid
B	Busy 32-bit TSS	Valid
C	32-bit call gate	Valid
D	Invalid	Invalid
E	32-bit trap gate	Invalid
F	32-bit interrupt gate	Invalid

-----

## Flags Affected

OF DF IF SF ZF AF PF CF

\*

The ZF flag is set unless the selector is invisible or of the wrong type, in which case the ZF flag is cleared.

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 6; the LAR instruction is unrecognized in Real Address Mode.

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode.

## Related Information

- Description
- Flags Affected
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

## LDS/LES/LFS/LGS/LSS-Load Full Pointer

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
----------	-------------	---	---	---	---	---	---	-------------

C5	/r	LDS	r16,m16:16	X X X X X X	DS:r16	` pointer from memory dword
C5	/r	LDS	r32,m16:32	X X X	DS:r32	` pointer from memory fword
C4	/r	LES	r16,m16:16	X X X X X X	ES:r16	` pointer from memory dword
C4	/r	LES	r32,m16:32	X X X	ES:r32	` pointer from memory fword
0F B4	/r	LFS	r16,m16:16	X X X	FS:r16	` pointer from memory dword
0F B4	/r	LFS	r32,m16:32	X X X	FS:r32	` pointer from memory fword
0F B5	/r	LGS	r16,m16:16	X X X	GS:r16	` pointer from memory dword
0F B5	/r	LGS	r32,m16:32	X X X	GS:r32	` pointer from memory fword
0F B2	/r	LSS	r16,m16:16	X X X	SS:r16	` pointer from memory dword
0F B2	/r	LSS	r32,m16:32	X X X	SS:r32	` pointer from memory fword

## Description

The LGS, LSS, LDS, LES, and LFS instructions read a full pointer from memory and store it in the selected segment register:register pair. The full pointer loads 16-bits into the segment register SS, DS, ES, FS, or GS. The other register loads 32-bits if the operand-size attribute is 32-bits, or loads 16-bits if the operand-size attribute is 16-bits. The other 16- or 32-bit register to be loaded is determined by the **r16** or **r32** register operand specified.

When an assignment is made to one of the segment registers, the descriptor is also loaded into the segment register. The data for the register is obtained from the descriptor table entry for the selector given.

A null selector (values 0000-0003) can be loaded into DS, ES, FS, or GS registers without causing a protection exception. (Any subsequent reference to a segment whose corresponding segment register is loaded with a null selector to address memory causes a #GP(0) exception. No memory reference to the segment occurs.)

The following is a listing of the Protected Mode checks and actions taken in the loading of a segment register:

IF SS is loaded;

```
IF selector is null THEN #GP(0); FI;
Selector index must be within its descriptor table limits ELSE
  #GP(selector);
Selector's RPL must equal CPL ELSE #GP(selector);
AR byte must indicate a writable data segment ELSE #GP(selector);
DPL in the AR byte must equal CPL ELSE #GP(selector);
Segment must be marked present ELSE #SS(selector);
Load SS with selector;
Load SS with descriptor;
```

IF DS, ES, FS, or GS is loaded with non-null selector:

```
Selector index must be within its descriptor table limits ELSE
  #GP(selector);
AR byte must indicate data or readable code segment ELSE
  #GP(selector);
IF data or nonconforming code
  THEN both the RPL and the CPL must be less than or equal to DPL in
    AR byte;
  ELSE #GP(selector);
Segment must be marked present ELSE #NP(selector);
Load segment register with selector and RPL bits;
Load segment register with descriptor;
```

IF DS, ES, FS or GS is loaded with a null selector:

```
Load segment register with selector;
Clear descriptor valid bit;
```

## Operation

```

CASE instruction OF
  LSS: Sreg is SS; (* Load SS register *)
  LDS: Sreg is DS; (* Load DS register *)
  LES: Sreg is ES; (* Load ES register *)
  LFS: Sreg is FS; (* Load FS register *)
  LGS: Sreg is GS; (* Load GS register *)
ESAC;
IF (OperandSize = 16)
THEN
  r16 `[Effective Address]; (* 16-bit transfer *)
  Sreg `[Effective Address + 2]; (* 16-bit transfer *)
  (* In Protected Mode, load the descriptor into the segment register *)
ELSE (* OperandSize = 32 *)
  r32 `[Effective Address]; (* 32-bit transfer *)
  Sreg `[Effective Address + 4]; (* 16-bit transfer *)
  (* In Protected Mode, load the descriptor into the segment register *)
FI;

```

---

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; the second operand must be a memory operand, not a register-if a register then #UD Fault; #GP(0) if a null selector is loaded into SS; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

The second operand must be a memory operand, not a register, Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## LEA-Load Effective Address

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
8D /r	LEA <i>r16</i> ,m	X	X	X	X	X	X	<i>r16</i> ` effective address for m
8D /r	LEA <i>r32</i> ,m		X	X	X			<i>r32</i> ` effective address for m

---

## Description

The LEA instruction calculates the effective address (offset part) and stores it in the specified register. The operand-size attribute of the instruction (represented by `OperandSize` in the algorithm under "Operation" above) is determined by the chosen register. The address-size attribute (represented by `AddressSize`) is determined by the attribute of the code segment. (See [Operand-Size and Address-Size Attributes](#).) The address-size and operand-size attributes affect the action performed by the LEA instruction, as follows:

OPERAND SIZE	ADDRESS SIZE	ACTION PERFORMED
16	16	16-bit effective address is calculated and stored in requested 16-bit register destination.
16	32	32-bit effective address is calculated. The lower 16-bits of the address are stored in the requested 16-bit register destination.
32	16	16-bit effective address is calculated. The 16-bit address is zero-extended and stored in the requested 32-bit register destination.
32	32	32-bit effective address is calculated and stored in the requested 32-bit register destination.

---

## Operation

```
IF OperandSize = 16 AND AddressSize = 16
THEN r16 ` Addr(m);
ELSE
  IF OperandSize = 16 AND AddressSize = 32
  THEN
    r16 ` Truncate_to_16bits(Addr(m)); (* 32-bit address *)
  ELSE
    IF OperandSize = 32 AND AddressSize = 16
    THEN
      r32 ` Truncate_to_16bits(Addr(m)) and zero extend;
```

```
ELSE
  IF OperandSize = 32 AND AddressSize = 32
    THEN r32 ← Addr(m);
  FI;
FI;
FI;
FI;
```

-----

## Protected Mode Exceptions

#UD if the second operand is a register.

-----

## Real Address Mode Exceptions

Interrupt 6 if the second operand is a register.

-----

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode.

-----

## Notes

Different assemblers may use different algorithms based on the size attribute and symbolic reference of the second operand.

-----

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

-----

## LEAVE-High Level Procedure Exit

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
C9	LEAVE		X	X	X	X	X	Set SP to BP, then pop BP
C9	LEAVE		X	X	X			Set ESP to EBP, then pop EBP

---

## Description

The LEAVE instruction reverses the actions of the ENTER instruction. By copying the frame pointer to the stack pointer, the LEAVE instruction releases the stack space used by a procedure for its local variables. The old frame pointer is popped into the BP or EBP register, restoring the caller's frame. A subsequent RET **nn** instruction removes any arguments pushed onto the stack of the exiting procedure.

---

## Operation

```
IF StackAddrSize = 16
THEN
  SP ← BP;
ELSE (* StackAddrSize = 32 *)
  ESP ← EBP;
FI;
IF OperandSize = 16
THEN
  BP ← Pop();
ELSE (* OperandSize = 32 *)
  EBP ← Pop();
FI;
```

---

## Protected Mode Exceptions

#SS(0) if the BP register does not point to a location within the limits of the current stack segment.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode.

---

## Related Information

- Description
- Flags Affected
- Operation
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

---

## LES-Load Full Pointer

See entry for LDS/LES/LFS/LGS/LSS.

---

## LFS-Load Full Pointer

See entry for LDS/LES/LFS/LGS/LSS.

---

## LGDT/LIDT-Load Global/Interrupt Descriptor Table Register

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F 01 /2	LGDT <a href="#">m16&amp;32</a>	X	X	X	X			GDTR `memory fword (6 bytes)
0F 01 /3	LIDT <a href="#">m16&amp;32</a>	X	X	X	X			IDTR `memory fword (6 bytes)

---

## Description

The LGDT and LIDT instructions load a linear base address and limit value from a six-byte data operand in memory into the GDTR or IDTR, respectively. If a 16-bit operand is used with the LGDT or LIDT instruction, the register is loaded with a 16-bit limit and a 24-bit base, and the high-order eight bits of the six-byte data operand are not used. If a 32-bit operand is used, a 16-bit limit and a 32-bit base is loaded; the high-order eight bits of the six-byte operand are used as high-order base address bits.

The SGDT and SIDT instructions always store into all 48 bits of the six-byte data operand. With the 16-bit processors, the upper eight bits are undefined after the SGDT or SIDT instruction is run. With the 32-bit processors, the upper right eight bits are written with the high-order eight address bits, for both a 16-bit operand and a 32-bit operand. If the LGDT or LIDT instruction is used with a 16-bit operand to load the register stored by the SGDT or SIDT instruction, the upper eight bits are stored as zeros.

The LGDT and LIDT instructions appear in operating system software; they are not used in application programs. They are the only instructions that directly load a linear address (i.e., not a segment relative address) in Protected Mode.

---

## Operation

```
IF instruction = LIDT
THEN
  IF OperandSize = 16
  THEN IDTR.Limit:Base ` m16:24 (* 24-bits of base loaded *)
  ELSE IDTR.Limit:Base ` m16:32
  FI;
ELSE (* instruction = LGDT *)
  IF OperandSize = 16
  THEN GDTR.Limit:Base ` m16:24 (* 24-bits of base loaded *)
  ELSE GDTR.Limit:Base ` m16:32;
  FI;
FI;
```

---

## Protected Mode Exceptions

#GP(0) if the current privilege level is not 0; #UD if the source operand is a register; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH; Interrupt 6 if the source operand is a register.

Note: These instructions are valid in Real Address Mode to allow power-up initialization for Protected Mode.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

# LGS-Load Full Pointer

See entry for LDS/LFS/LGS/LSS.

---

# LLDT-Load Local Descriptor Table Register

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F 00 /2	LLDT <a href="#">r/m16</a>					<a href="#">X</a>	<a href="#">X</a>	LDTR <code>`r/m16 selector</code>

---

## Description

The LLDT instruction loads the Local Descriptor Table register (LDTR). The word operand (memory or register) to the LLDT instruction should contain a selector to the Global Descriptor Table (GDT). The GDT entry should be a Local Descriptor Table. If so, then the LDTR is loaded from the entry. The descriptor registers DS, ES, SS, FS, GS, and CS are not affected. The LDT field in the task state segment does not change.

The selector operand can be 0; if so, the LDTR is marked invalid. All descriptor references (except by the LAR, VERR, VERW or LSL instructions) cause a #GP fault.

The LLDT instruction is used in operating system software; it is not used in application programs.

---

## Operation

LDTR ``SRC`;

# Protected Mode Exceptions

#GP(0) if the current privilege level is not 0; #GP(selector) if the selector operand does not point into the Global Descriptor Table, or if the entry in the GDT is not a Local Descriptor Table; #NP(selector) if the LDT descriptor is not present; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault.

---

## Real Address Mode Exceptions

Interrupt 6; the LLDT instruction is not recognized in Real Address Mode.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode (because the instruction is not recognized, it will not run or perform a memory reference).

### Note

The operand-size attribute has no effect on this instruction.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## LIDT-Load Interrupt Descriptor Table Register

See entry for LGDT/LIDT-Load Global Descriptor Table Register/Load Interrupt Descriptor Table Register.

---

## LMSW-Load Machine Status Word

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F 01 /6	LMSW <i>r/m16</i>					X	X	Load <i>r/m16</i> into machine status word

## Description

The LMSW instruction loads the machine status word (part of the CR0 register) from the source operand. This instruction can be used to switch to Protected Mode; if so, it must be followed by an intrasegment jump to flush the instruction queue. The LMSW instruction will not switch back to Real Address Mode.

The LMSW instruction is used only in operating system software. It is not used in application programs.

## Operation

MSW ← *r/m16*; (\* 16 bits is stored in the machine status word \*)

## Protected Mode Exceptions

#GP(0) if the current privilege level is not 0; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

## Virtual 8086 Mode Exceptions

Same exceptions as in Protected Mode.

## Notes

The operand-size attribute has no effect on this instruction. This instruction is provided for compatibility with the Intel286 processor; programs for the Intel386, Intel486, and Pentium processors should use the MOV CR0, ... instruction instead. The LMSW instruction does not affect the PG, ET, or NE bits, and it cannot be used to clear the PE bit.

-----

# Related Information

- Description
- Flags Affected
- Notes
- Operation
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

-----

# LOCK-Assert LOCK# Signal Prefix

-----

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
F0	LOCK	X	X	X	X	X	X	Assert LOCK# signal for next instruction

-----

# Description

The LOCK prefix causes the LOCK# signal of the processor to be asserted during processing of the instruction that follows it. In a multiprocessor environment, this signal can be used to ensure that the processor has exclusive use of any shared memory while LOCK# is asserted. The read-modify-write sequence typically used to implement test-and-set on the Pentium processor is the BTS instruction.

The LOCK prefix functions only with the following instructions:

BTS, BTR, BTC	mem, reg/imm
XCHG	reg, mem
XCHG	mem, reg
ADD, OR, ADC, SBB, AND, SUB, XOR	mem, reg/imm
NOT, NEG, INC, DEC	mem
CMPXCHG, XADD	

An undefined opcode trap will be generated if a LOCK prefix is used with any instruction not listed above.

The integrity of the LOCK prefix is not affected by the alignment of the memory field. Memory locking is observed for arbitrarily misaligned fields.

#UD if the LOCK prefix is used with an instruction not listed in the "Description" section above; other exceptions can be generated by the subsequent (locked) instruction.

Interrupt 6 if the LOCK prefix is used with an instruction not listed in the "Description" section above; exceptions can still be generated by the subsequent (locked) instruction.

#UD if the LOCK prefix is used with an instruction not listed in the "Description" section above; exceptions can still be generated by the subsequent (locked) instruction.

## Virtual 8086 Mode Exceptions

## Details Table

Encoding	Instruction		0	1	2	3	4	5	Description
AC	LDS m8	X	X	X	X	X	X	AL	' byte at [(E)SI]

AD	LODS <b>m16</b>	<b>X X X X X X</b> AX ` word at [(E)SI]
AD	LODS <b>m32</b>	<b>X X X</b> EAX ` dword at [(E)SI]
AC	LODSB	<b>X X X X X X</b> AL ` byte at DS:[(E)SI]
AD	LODSW	<b>X X X X X X</b> AX ` word at DS:[(E)SI]
AD	LODSD	<b>X X X</b> EAX ` dword at DS:[(E)SI]

## Description

The LODS instruction loads the AL, AX, or EAX register with the memory byte, word, or doubleword at the location pointed to by the source-index register. After the transfer is made, the source-index register is automatically advanced. If the DF flag is 0 (the CLD instruction was run), the source index increments; if the DF flag is 1 (the STD instruction was run), it decrements. The increment or decrement is 1 if a byte is loaded, 2 if a word is loaded, or 4 if a doubleword is loaded.

If the address-size attribute for this instruction is 16-bits, the SI register is used for the source-index register; otherwise the address-size attribute is 32-bits, and the ESI register is used. The address of the source data is determined solely by the contents of the ESI or SI register. Load the correct index value into the SI register before running the LODS instruction. The LODSB, LODSW, and LODSD instructions are synonyms for the byte, word, and doubleword LODS instructions.

The LODS instruction can be preceded by the REP prefix; however, the LODS instruction is used more typically within a LOOP construct, because further processing of the data moved into the EAX, AX, or AL register is usually necessary.

## Operation

```

AddressSize = 16
THEN use SI for source-index
ELSE (* AddressSize = 32 *)
    use ESI for source-index;
FI;
IF byte type of instruction
THEN
    AL `[source-index]; (* byte load *)
    IF DF = 0 THEN IncDec ` 1 ELSE IncDec ` -1; FI;
ELSE
    IF OperandSize = 16
    THEN
        AX `[source-index]; (* word load *)
        IF DF = 0 THEN IncDec ` 2 ELSE IncDec ` -2; FI;
    ELSE (* OperandSize = 32 *)
        FAX `[source-index]; (* dword load *)
        IF DF = 0 THEN IncDec ` 4 ELSE IncDec ` -4; FI;
    FI;
FI;
source-index ` source-index + IncDec

```

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

# Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Related Information

- [Description](#)
- [Flags Affected](#)
- [Operation](#)
- [Protected Mode Exceptions](#)
- [Real Address Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

## LOOP/LOOPcond-Loop Control with CX Counter

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
E2 cb	LOOP <a href="#">rel8</a>	X	X	X	X	X	X	DEC (E)CX; jump if (E)CX <> 0
E2 cb	LOOPW <a href="#">rel8</a>			X	X	X		DEC CX; jump if CX <> 0
E2 cb	LOOPD <a href="#">rel8</a>			X	X	X		DEC ECX; jump if ECX <> 0
E1 cb	LOOPE <a href="#">rel8</a>	X	X	X	X	X	X	DEC (E)CX, jump if (E)CX <> 0 and ZF=1
E1 cb	LOOPEW <a href="#">rel8</a>			X	X	X		DEC CX, jump if CX <> 0 and ZF=1
E1 cb	LOOPED <a href="#">rel8</a>			X	X	X		DEC ECX; jump if ECX <> 0 and ZF=1
E1 cb	LOOPZ <a href="#">rel8</a>	X	X	X	X	X	X	DEC (E)CX; jump if (E)CX <> 0 and ZF=1
E1 cb	LOOPZW <a href="#">rel8</a>			X	X	X		DEC CX; jump if CX <> 0 and ZF=1
E1 cb	LOOPZD <a href="#">rel8</a>			X	X	X		DEC ECX; jump if ECX <> 0 and ZF=1

E0 cb	LOOPNE <i>rel8</i>	X X X X X X	DEC (E)CX; jump if (E)CX <> 0 and ZF=0
E0 cb	LOOPNEW <i>rel8</i>	X X X	DEC CX; jump if CX <> 0 and ZF=0
E0 cb	LOOPNED <i>rel8</i>	X X X	DEC ECX; jump if ECX <> 0 and ZF=0
E0 cb	LOOPNZ <i>rel8</i>	X X X X X X	DEC (E)CX; jump if (E)CX <> 0 and ZF=0
E0 cb	LOOPNZW <i>rel8</i>	X X X	DEC CX; jump if CX <> 0 and ZF=0
E0 cb	LOOPNZD <i>rel8</i>	X X X	DEC ECX; jump if ECX <> 0 and ZF=0

-----

## Description

The LOOP instruction decrements the count register without changing any of the flags. Conditions are then checked for the form of the LOOP instruction being used. If the conditions are met, a short jump is made to the label given by the operand to the LOOP instruction. If the address-size attribute is 16-bits, the CX register is used as the count register; otherwise the ECX register is used. The operand of the LOOP instruction must be in the range from 128 (decimal) bytes before the instruction to 127 bytes ahead of the instruction.

The LOOP instructions provide iteration control and combine loop index management with conditional branching. Use the LOOP instruction by loading an unsigned iteration count into the count register, then code the LOOP instruction at the end of a series of instructions to be iterated. The destination of the LOOP instruction is a label that points to the beginning of the iteration.

-----

## Operation

IF AddressSize = 16 THEN CountReg is CX ELSE CountReg is ECX; FI;  
CountReg ` CountReg - 1;

IF instruction<> LOOP  
THEN  
IF (instruction = LOOPE) OR (instruction = LOOPZ)  
THEN BranchCond ` (ZF = 1) AND (CountReg <> 0);  
FI;  
IF (instruction = LOOPNE) OR (instruction = LOOPNZ)  
THEN BranchCond ` (ZF = 0) AND (CountReg <> 0);  
FI;  
FI;

IF BranchCond  
THEN  
IF OperandSize = 16  
THEN  
IP ` IP + SignExtend(*rel8*);  
ELSE (\* OperandSize = 32 \*)  
EIP ` EIP + SignExtend(*rel8*);  
FI;  
FI;

-----

## Protected Mode Exceptions

#GP(0) if the offset jumped to is beyond the limits of the current code segment.

-----

## Notes

The unconditional LOOP instruction takes longer to run than a two-instruction sequence, which decrements the counter register and jumps if the count does not equal zero.

All branches are converted into 16-byte code fetches regardless of jump address or cacheability.

## Related Information

- Description
- Flags Affected
- Notes
- Operation
- Protected Mode Exceptions
- Protected Mode Exceptions
- Virtual 8086 Mode Exceptions

## LSL-Load Segment Limit

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F 03 /r	LSL r16,r/m16	X	X	X	X			r16 ` byte granular segment limit for selector r/m16
0F 03 /r	LSL r32,r/m32		X	X	X			r32 ` byte granular segment limit for selector r/m32
0F 03 /r	LSL r16,r/m16	X	X	X	X			r16 ` page granular segment limit for selector r/m16
0F 03 /r	LSL r32,r/m32		X	X	X			r32 ` page granular segment limit for selector r/m32

## Description

The LSL instruction loads a register with an unscrambled segment limit, and sets the ZF flag, provided that the source selector is visible at the current privilege level and RPL, within the descriptor table, and that the descriptor is a type accepted by the LSL instruction. Otherwise, the ZF flag is cleared, and the destination register is unchanged. The segment limit is loaded as a byte granular value. If the descriptor has a page granular segment limit, the LSL instruction will translate it to a byte limit before loading it in the destination register (shift left 12 the 20-bit "raw" limit from descriptor, then OR with 00000FFFH).

The 32-bit forms of the LSL instruction store the 32-bit byte granular limit in the 32-bit destination register. For 16-bit operand sizes, the limit is computed to form a valid 32-bit limit. However, the upper 16 bits are chopped and only the low-order 16 bits are loaded into the destination operand.

Code and data segment descriptors are valid for the LSL instruction.

The valid special segment and gate descriptor types for the LSL instruction are given in the following table:

TYPE	NAME	VALID/INVALID
0	Invalid	Invalid
1	Available 16-bit TSS	Valid
2	LDT	Valid
3	Busy 16-bit TSS	Valid
4	16-bit call gate	Invalid
5	16-bit/32-bit task gate	Invalid
6	16-bit trap gate	Invalid
7	16-bit interrupt gate	Invalid
8	Invalid	Invalid
9	Available 32-bit TSS	Valid
A	Invalid	Invalid
B	Busy 32-bit TSS	Valid
C	32-bit call gate	Invalid
D	Invalid	Invalid
E	32-bit trap gate	Invalid
F	32-bit interrupt gate	Invalid

-----

## Flags Affected

OF DF IF SF ZF AF PF CF

\*

The ZF flag is set unless the selector is invisible or of the wrong type, in which case the ZF flag is cleared.

-----

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Real Address Mode Exceptions

Interrupt 6; the LSL instruction is not recognized in Real Address Mode.

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode (because the instruction is not recognized, it will not run or perform a memory reference).

## Related Information

- Description
- Flags Affected
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

## LSS-Load Full Pointer

See entry for LDS/LES/LFS/LGS/LSS.

## LTR-Load Task Register

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F 00 /3	LTR <i>r/m16</i>		X	X	X	X		Load EA word into task register

## Description

The LTR instruction loads the task register with a selector from the source register or memory location specified by the operand. The loaded TSS is marked busy. A task switch does not occur.

The LTR instruction is used only in operating system software; it is not used in application programs.

---

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #GP(0) if the current privilege level is not 0; #GP(selector) if the object named by the source selector is not a TSS or is already busy; #NP(selector) if the TSS is marked "not present;" #PF(fault-code) for a page fault.

---

## Real Address Mode Exceptions

Interrupt 6; the LTR instruction is not recognized in Real Address Mode.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode.

---

## Notes

The operand-size attribute has no effect on this instruction.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## MOV-Move Data

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
88 /r	MOV r/m8,r8	X	X	X	X	X	X	Move byte register to r/m byte
89 /r	MOV r/m16,r16	X	X	X	X	X	X	Move word register to r/m word
89 /r	MOV r/m32,r32			X	X	X		Move dword register to r/m dword
8A /r	MOV r8,r/m8	X	X	X	X	X	X	Move r/m byte to byte register
8B /r	MOV r16,r/m16	X	X	X	X	X	X	Move r/m word to word register
8B /r	MOV r32,r/m32			X	X	X		Move r/m dword to dword register
8C /r	MOV r/m16,Sreg	X	X	X	X	X	X	Move segment register to r/m word
8E /r	MOV Sreg,r/m16	X	X	X	X	X	X	Move r/m word to segment register
A0	MOV AL,moffs8	X	X	X	X	X	X	Move byte at (seg:offset) to AL
A1	MOV AX,moffs16	X	X	X	X	X	X	Move word at (seg:offset) to AX
A1	MOV EAX,moffs32			X	X	X		Move dword at (seg:offset) to EAX
A2	MOV moffs8,AL	X	X	X	X	X	X	Move AL to (seg:offset)
A3	MOV moffs16,AX	X	X	X	X	X	X	Move AX to (seg:offset)
A3	MOV moffs32,EAX			X	X	X		Move EAX to (seg:offset)
B0+rb	MOV r8,imm8	X	X	X	X	X	X	Move immediate byte to byte register
B8+rw	MOV r16,imm16	X	X	X	X	X	X	Move immediate word to word register
B8+rd	MOV r32,imm32			X	X	X		Move immediate dword to dword register
C6 /0	MOV r/m8,imm8	X	X	X	X	X	X	Move immediate byte to r/m byte
C7 /0	MOV r/m16,imm16	X	X	X	X	X	X	Move immediate word to r/m word
C7 /0	MOV r/m32,imm32			X	X	X		Move immediate dword to r/m dword

## Description

The MOV instruction copies the second operand to the first operand.

If the destination operand is a segment register (DS, ES, SS, etc.), then data from a descriptor is also loaded into the shadow portion of the register. The data for the register is obtained from the descriptor table entry for the selector given. A null selector (values 0000-0003) can be loaded into the DS, ES, FS, and GS registers without causing an exception; however, use of these registers causes a #GP(0) exception, and no memory reference occurs.

A MOV into SS instruction inhibits all interrupts until after the processing of the next instruction (which should be a MOV into ESP instruction).

Loading a segment register under Protected Mode results in special checks and actions, as described in the following listing:

```

IF SS is loaded;
THEN
  IF selector is null THEN #GP(0);
  Selector index must be within its descriptor table limits else #GP(selector);
  Selector's RPL must equal CPL else #GP(selector);
  AR byte must indicate a writable data segment else #GP(selector);
  DPL in the AR byte must equal CPL else #GP(selector);
  Segment must be marked present else #SS(selector);
  Load SS with selector;
  Load SS with descriptor.
FI;
IF DS, ES, FS or GS is loaded with non-null selector;
```

```
THEN
  Selector index must be within its descriptor table limits
  else #GP(selector);
  AR byte must indicate data or readable code segment else #GP(selector);
  IF data or nonconforming code segment
  THEN both the RPL and the CPL must be less than or equal to DPL in AR byte;
  ELSE #GP(selector);
  FI;
  Segment must be marked present else #NP(selector);
  Load segment register with selector;
  Load segment register with descriptor;
  FI;
  IF DS, ES, FS or GS is loaded with a null selector;
  THEN
    Load segment register with selector;
    Clear descriptor valid bit;
  FI;
```

---

## Operation

```
DEST ← SRC;
```

---

## Protected Mode Exceptions

#GP, #SS, and #NP if a segment register is being loaded; otherwise, #GP(0) if the destination is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

# MOV-Move to/from Control Registers

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F 22 /r	MOV CR0,r32				X	X	X	Move r32 to control register 0
0F 22 /r	MOV CR2,r32				X	X	X	Move r32 to control register 2
0F 22 /r	MOV CR3,r32				X	X	X	Move r32 to control register 3
0F 22 /r	MOV CR4,r32					X		Move r32 to control register 4
0F 20 /r	MOV r32,CR0				X	X	X	Move control register 0 to r32
0F 20 /r	MOV r32,CR2				X	X	X	Move control register 2 to r32
0F 20 /r	MOV r32,CR3				X	X	X	Move control register 3 to r32
0F 20 /r	MOV r32,CR4					X		Move control register 4 to r32

## Description

The above forms of the MOV instruction store or load CR0, CR2, CR3, and CR4 to or from a general purpose register. Thirty-two bit operands are always used with these instructions, regardless of the operand-size attribute.

## Operation

DEST ← SRC;

## Flags Affected

OF DF IF SF ZF AF PF CF

? ? ? ? ?

The OF, SF, ZF, AF, PF, and CF flags are undefined.

---

## Protected Mode Exceptions

#GP(0) if the current privilege level is not 0. #GP(0) if an attempt is made to write a 1 to any reserved bits of CR4.

---

## Real Address Mode Exceptions

Interrupt 13 if an attempt is made to write a 1 to any reserved bits of CR4.

---

## Virtual 8086 Mode Exceptions

#GP(0) if instruction processing is attempted.

---

## Notes

The *reg* field within the ModR/M byte specifies which of the special registers in each category is involved. The two bits in the *mod* field are always 11. The *rm* field specifies the general register involved.

Always set undefined or reserved bits to the value previously read.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## MOV-Move to/from Debug Registers

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F 21 /r	MOV <b>r32</b> ,DR0-3				X	X	X	Move debug register (0,1,2, or 3) to r32
0F 21 /r	MOV <b>r32</b> ,DR4/DR5					X		Move debug register (4 or 5) to r32
0F 21 /r	MOV <b>r32</b> ,DR6/DR7				X	X	X	Move debug register (6 or 7) to r32
0F 23 /r	MOV DR0-DR3, <b>r32</b>	X	X	X	X			Move r32 to debug register (0,1,2, or 3)
0F 23 /r	MOV DR4/DR5, <b>r32</b>					X		Move r32 to debug register (4 or 5)
0F 23 /r	MOV DR6/DR7, <b>r32</b>	X	X	X	X			Move r32 to debug register (6 or 7)

---

## Description

The above forms of the MOV instruction store or load the DR0, DR1, DR2, DR3, DR6 and DR7 debug registers to or from a general purpose register.

Thirty-two bit operands are always used with these instructions, regardless of the operand-size attribute.

When the DE (Debug Extension) bit in CR4 is clear, MOV instructions using debug registers operate in a manner that is compatible with Intel386 and Intel486 CPUs. References to DR4 and DR5 refer to DR6 and DR7, respectively. When the DE bit in CR4 is set, attempts to run MOV instructions using DR4 and DR5 result in an Undefined Opcode (#UD) exception.

---

## Operation

```
IF ((DE = 1) and (SRC or DEST = DR4 or DR5))
THEN
    #UD;
ELSE
    DEST ← SRC;
```

---

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
?		?	?	?	?	?	

The OF, SF, ZF, AF, PF, and CF flags are undefined.

---

## Protected Mode Exceptions

#GP(0) if the current privilege level is not 0. #UD if the DE (Debug Extensions) bit of CR4 is set and a MOV instruction is run using DR4 or DR5.

---

## Real Address Mode Exceptions

#GP(0) if an attempt is made to write a 1 to any reserved bits of CR4. #UD if the DE (Debug Extensions) bit of CR4 is set and a MOV instruction is run using DR4 or DR5.

---

## Virtual 8086 Mode Exceptions

#GP(0) if instruction processing is attempted.

---

## Notes

The instructions must be run at privilege level 0 or in real-address mode; otherwise, a protection exception will be raised.

The *reg* field within the ModR/M byte specifies which of the special registers in each category is involved. The two bits in the *mod* field are always 11. The *r/m* field specifies the general register involved.

Always set undefined or reserved bits to the value previously read.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## MOV-Move to/from Test Registers

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F 24 /r	MOV r32,TR3					X		Move test register (3) to r32
0F 24 /r	MOV r32,TR4/TR5					X		Move test register (4 or 5) to r32
0F 24 /r	MOV r32,TR6/TR7			X	X			Move test register (6 or 7) to r32
0F 26 /r	MOV TR3,r32					X		Move r32 to test register (3)
0F 26 /r	MOV TR4/TR5,r32					X		Move r32 to test register (4 or 5)
0F 26 /r	MOV TR6/TR7,r32			X	X			Move r32 to test register (6 or 7)

---

## Description

The above forms of the MOV instruction store or load TR3, TR4, TR5, TR6, and TR7 to or from a general purpose register.

Thirty-two bit operands are always used with these instructions, regardless of the operand-size attribute.

---

## Operation

DEST`SRC;

---

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
?			?	?	?	?	?

The OF, SF, ZF, AF, PF, and CF flags are undefined.

---

## Protected Mode Exceptions

#GP(0) if the current privilege level is not 0. #GP(0) if an attempt is made to write a 1 to any reserved bits of CR4.

---

## Real Address Mode Exceptions

Interrupt 13 if an attempt is made to write a 1 to any reserved bits of CR4.

---

## Virtual 8086 Mode Exceptions

#GP(0) if instruction processing is attempted.

---

## Notes

The instructions must be run at privilege level 0 or in real-address mode; otherwise, a protection exception will be raised.

The *reg* field within the ModR/M byte specifies which of the special registers in each category is involved. The two bits in the *mod* field are always 11. The *rm* field specifies the general register involved.

Always set undefined or reserved bits to the value previously read.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## MOVS/MOVSb/MOVSx/MOVSd-Move Data from String to String

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
A4	MOVS <b>m8,m8</b>	X	X	X	X	X	X	Move byte [(E)SI] to ES:[(E)DI]
A5	MOVS <b>m16,m16</b>	X	X	X	X	X	X	Move word [(E)SI] to ES:[(E)DI]
A5	MOVS <b>m32,m32</b>				X	X	X	Move dword [(E)SI] to ES:[(E)DI]
A4	MOVSB	X	X	X	X	X	X	Move byte DS:[(E)SI] to ES:[(E)DI]
A5	MOVSW	X	X	X	X	X	X	Move word DS:[(E)SI] to ES:[(E)DI]
A5	MOVSD				X	X	X	Move dword DS:[(E)SI] to ES:[(E)DI]

## Description

The MOVS instruction copies the byte or word at [(E)SI] to the byte or word at ES:[(E)DI]. The destination operand must be addressable from the ES register; no segment override is possible for the destination. A segment override can be used for the source operand; the default is the DS register.

The addresses of the source and destination are determined solely by the contents of the (E)SI and (E)DI registers. Load the correct index values into the (E)SI and (E)DI registers before running the MOVS instruction. The MOVSB, MOVSW, and MOVSD instructions are synonyms for the byte, word, and doubleword MOVS instructions.

After the data is moved, both the (E)SI and (E)DI registers are advanced automatically. If the DF flag is 0 (the CLD instruction was run), the registers are incremented; if the DF flag is 1 (the STD instruction was run), the registers are decremented. The registers are incremented or decremented by 1 if a byte was moved, 2 if a word was moved, or 4 if a doubleword was moved.

The MOVS instruction can be preceded by the REP prefix for block movement of ECX bytes or words. Refer to the REP instruction for details of this operation.

## Operation

```

IF (instruction = MOVSD) OR (instruction has doubleword operands)
THEN OperandSize ` 32;
ELSE OperandSize ` 16;
IF AddressSize = 16
THEN use SI for source-index and DI for destination-index;
ELSE (* AddressSize = 32 *)
    use ESI for source-index and EDI for destination-index;
FI;
IF byte type of instruction
THEN
    [destination-index] ` [source-index]; (* byte assignment *)
    IF DF = 0 THEN IncDec ` 1 ELSE IncDec ` -1; FI;
ELSE
    IF OperandSize = 16
    THEN
        [destination-index] ` [source-index]; (* word assignment *)
        IF DF = 0 THEN IncDec ` 2 ELSE IncDec ` -2; FI;
    ELSE (* OperandSize = 32 *)
        [destination-index] ` [source-index]; (* doubleword assignment *)
        IF DF = 0 THEN IncDec ` 4 ELSE IncDec ` -4; FI;
    FI;
FI;
source-index ` source-index + IncDec;
destination-index ` destination-index + IncDec;

```

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Related Information

- [Description](#)
- [Flags Affected](#)
- [Operation](#)
- [Protected Mode Exceptions](#)
- [Real Address Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

## MOVSX-Move with Sign-Extend

### Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F BE /r	MOVSX r16,r/m8	X	X	X				r16 ` sign-extended r/m byte
0F BE /r	MOVSX r32,r/m8	X	X	X				r32 ` sign-extended r/m byte
0F BF /r	MOVSX r32,r/m16	X	X	X				r32 ` sign-extended r/m word

# Description

The MOVZX instruction reads the contents of the effective address or register as a byte or a word, sign-extends the value to the operand-size attribute of the instruction (16 or 32 bits), and stores the result in the destination register.

---

# Operation

DEST ← SignExtend(SRC);

---

# Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

# Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

# Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

# Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

# MOVZX-Move with Zero-Extend

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F B6 /r	MOVZX <a href="#">r16,r/m8</a>				<a href="#">X</a>	<a href="#">X</a>	<a href="#">X</a>	r16 ` zero-extended r/m byte
0F B6 /r	MOVZX <a href="#">r32,r/m8</a>				<a href="#">X</a>	<a href="#">X</a>	<a href="#">X</a>	r32 ` zero-extended r/m byte
0F B7 /r	MOVZX <a href="#">r32,r/m16</a>				<a href="#">X</a>	<a href="#">X</a>	<a href="#">X</a>	r32 ` zero-extended r/m word

---

## Description

The MOVZX instruction reads the contents of the effective address or register as a byte or a word, zero extends the value to the operand-size attribute of the instruction (16 or 32 bits), and stores the result in the destination register.

---

## Operation

DEST `ZeroExtend(SRC);

---

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same Exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

# MUL-Unsigned Multiplication of AL, AX, or EAX

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
F6 /r	MUL <a href="#">r/m8</a>	X	X	X	X	X	X	Unsigned multiply (AX ← AL * r/m byte)
F7 /4	MUL <a href="#">r/m16</a>	X	X	X	X	X	X	Unsigned multiply (DX:AX ← AX * r/m word)
F7 /4	MUL <a href="#">r/m32</a>		X	X	X	X		Unsigned multiply (EDX:EAX ← EAX * r/m dword)

---

## Description

The MUL instruction performs unsigned multiplication. Its actions depend on the size of its operand, as follows:

- A byte operand is multiplied by the AL value; the result is left in the AX register. The CF and OF flags are cleared if the AH value is 0; otherwise, they are set.
  - A word operand is multiplied by the AX value; the result is left in the DX:AX register pair. The DX register contains the high-order 16-bits of the product. The CF and OF flags are cleared if the DX value is 0; otherwise, they are set.
  - A doubleword operand is multiplied by the EAX value and the result is left in the EDX:EAX register. The EDX register contains the high-order 32-bits of the product. The CF and OF flags are cleared if the EDX value is 0; otherwise, they are set.
- 

## Operation

IF byte-size operation  
THEN AX ← AL \* *r/m8*

```

ELSE (* word or doubleword operation *)
  IF OperandSize = 16
    THEN DX:AX ` AX * r/m16
  ELSE (* OperandSize = 32 *)
    EDX:EAX ` EAX * r/m32
  FI;
FI;

```

-----

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
*			?	?	?	?	*

The OF and CF flags are cleared if the upper half of the result is 0; otherwise they are set; the SF, ZF, AF, and PF flags are undefined.

-----

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

-----

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

# NEG-Two's Complement Negation

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
F6 /3	NEG <i>r/m8</i>	X	X	X	X	X	X	Two's complement negate <i>r/m</i> byte
F7 /3	NEG <i>r/m16</i>	X	X	X	X	X	X	Two's complement negate <i>r/m</i> word
F7 /3	NEG <i>r/m32</i>			X	X	X		Two's complement negate <i>r/m</i> dword

---

## Description

The NEG instruction replaces the value of a register or memory operand with its two's complement. The operand is subtracted from zero, and the result is placed in the operand.

The CF flag is set, unless the operand is zero, in which case the CF flag is cleared.

---

## Operation

IF *r/m* = 0 THEN CF ← 0 ELSE CF ← 1; FI;  
*r/m* ← - *r/m*

---

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
*			*	*	*	*	*

The CF flag is set unless the operand is zero, in which case the CF flag is cleared; the OF, SF, ZF, and PF flags are set according to the result.

---

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in real-address mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Related Information

- [Description](#)
- [Flags Affected](#)
- [Operation](#)
- [Protected Mode Exceptions](#)
- [Real Address Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

---

## NOP-No Operation

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
90	NOP	X	X	X	X	X	X	No operation

---

## Description

The NOP instruction performs no operation. The NOP instruction is a one-byte instruction that takes up space but affects none of the machine context except the (E)IP register.

The NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Protected Mode Exceptions](#)

[Protected Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## NOT-One's Complement Negation

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
F6 /2	NOT <a href="#">r/m8</a>	X	X	X	X	X	X	Reverse each bit of r/m byte
F7 /2	NOT <a href="#">r/m16</a>	X	X	X	X	X	X	Reverse each bit of r/m word
F7 /2	NOT <a href="#">r/m32</a>				X	X	X	Reverse each bit of r/m dword

---

## Description

The NOT instruction inverts the operand; every 1 becomes a 0, and vice versa.

---

## Operation

$r/m \leftarrow \text{NOT } r/m$

---

# Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

# Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

# Virtual 8086 Mode Exceptions

Same exceptions as in real-address mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

# Related Information

- [Description](#)
- [Flags Affected](#)
- [Operation](#)
- [Protected Mode Exceptions](#)
- [Real Address Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

# OR-Logical Inclusive OR

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0C ib	OR AL,imm8	X	X	X	X	X	X	OR immediate byte to AL
0D iw	OR AX,imm16	X	X	X	X	X	X	OR immediate word to AX
0D id	OR EAX,imm32					X	X X X	OR immediate dword to EAX
80 /1 ib	OR r/m8,imm8	X	X	X	X	X	X	OR immediate byte to r/m byte

81	/1 iw	OR <code>r/m16,imm16</code>	X X X X X X	OR immediate word to r/m word
81	/1 id	OR <code>r/m32,imm32</code>	X X X	OR immediate dword to r/m dword
83	/1 ib	OR <code>r/m16,imm8</code>	X X X	OR sign-extended immediate byte with r/m word
81	/1 ib	OR <code>r/m32,imm8</code>	X X X	OR sign-extended immediate byte with r/m dword
08	/r	OR <code>r/m8,r8</code>	X X X X X X	OR byte register to r/m byte
09	/r	OR <code>r/m16,r16</code>	X X X X X X	OR word register to r/m word
09	/r	OR <code>r/m32,r32</code>	X X X	OR dword register to r/m dword
0A	/r	OR <code>r8,r/m8</code>	X X X X X X	OR byte register to r/m byte
0B	/r	OR <code>r16,r/m16</code>	X X X X X X	OR word register to r/m word
0B	/r	OR <code>r32,r/m32</code>	X X X	OR dword register to r/m dword

## Description

The OR instruction computes the inclusive OR of its two operands and places the result in the first operand. Each bit of the result is 0 if both corresponding bits of the operands are 0; otherwise, each bit is 1.

## Operation

DEST ← DEST OR SRC;  
CF ← 0;  
OF ← 0

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
0			*	*	?	*	0

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result; the AF flag is undefined.

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

# Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

# Virtual 8086 Mode Exceptions

Same exceptions as in real-address mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

# Related Information

- Description
- Flags Affected
- Operation
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

# OUT-Output to Port

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
E6 ib	OUT imm8,AL	X	X	X	X	X	X	Output byte from AL to port imm8
E7 ib	OUT imm8,AX	X	X	X	X	X	X	Output word from AX to port imm8
E7 ib	OUT imm8,EAX				X	X	X	Output dword from EAX to port imm8
EE	OUT DX,AL	X	X	X	X	X	X	Output byte from AL to port number in DX
EF	OUT DX,AX	X	X	X	X	X	X	Output word from AX to port number in DX
EF	OUT DX,EAX				X	X	X	Output dword from EAX to port number in DX

---

## Description

The OUT instruction transfers a data byte or data word from the register (AL, AX, or EAX) given as the second operand to the output port numbered by the first operand. Output to any port from 0 to 65535 is performed by placing the port number in the DX register and then using an OUT instruction with the DX register as the first operand. If the instruction contains an eight-bit port ID, that value is zero-extended to 16-bits.

---

## Operation

```
IF (PE = 1) AND ((VM = 1) OR (CPL > IOPL))
THEN (* Virtual 8086 mode, or protected mode with CPL > IOPL *)
  IF NOT I/O-Permission (DEST, width(DEST))
  THEN #GP(0);
  FI;
FI;
[DEST] ← SRC; (* I/O address space used *)
```

---

## Protected Mode Exceptions

#GP(0) if the current privilege level is higher (has less privilege) than the I/O privilege level and any of the corresponding I/O permission bits in the TSS equals 1.

---

## Virtual 8086 Mode Exceptions

#GP(0) fault if any of the corresponding I/O permission bits in the TSS equals 1.

---

## Notes

After the OUT or OUTS instructions are run, the Pentium processor ensures that the EWBE# has been sampled active before beginning to run the next instruction. Note that the instruction may be prefetched if EWBE# is not active, but it will not run until EWBE# is sampled active.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

# OUTS/OUTSB/OUTSW/OUTSD-Output String to Port

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
6E	OUTS DX,r/m8	X	X	X	X	X		Output byte [(E)SI] to port in DX
6F	OUTS DX,r/m16	X	X	X	X	X		Output word [(E)SI] to port in DX
6F	OUTS DX,r/m32			X	X	X		Output dword [(E)SI] to port in DX
6E	OUTSB	X	X	X	X	X		Output byte [DS:(E)SI] to port in DX
6F	OUTSW	X	X	X	X	X		Output word [DS:(E)SI] to port in DX
6F	OUTSD			X	X	X		Output dword [DS:(E)SI] to port in DX

## Description

The OUTS instruction transfers data from the memory byte, word, or doubleword at the source-index register to the output port addressed by the DX register. If the address-size attribute for this instruction is 16-bits, the SI register is used for the source-index register; otherwise, the address-size attribute is 32-bits, and the ESI register is used for the source-index register.

The OUTS instruction does not allow specification of the port number as an immediate value. The port must be addressed through the DX register value. Load the correct value into the DX register before running the OUTS instruction.

The address of the source data is determined by the contents of source-index register. Load the correct index value into the SI or ESI register before running the OUTS instruction.

After the transfer, source-index register is advanced automatically. If the DF flag is 0 (the CLD instruction was run), the source-index register is incremented; if the DF flag is 1 (the STD instruction was run), it is decremented. The amount of the increment or decrement is 1 if a byte is output, 2 if a word is output, or 4 if a doubleword is output.

The OUTSB, OUTSW, and OUTSD instructions are synonyms for the byte, word, and doubleword OUTS instructions. The OUTS instruction can be preceded by the REP prefix for block output of ECX bytes or words. Refer to the REP instruction for details on this operation.

## Operation

IF AddressSize = 16  
THEN use SI for source-index;

```

ELSE (* AddressSize = 32 *)
  use ESI for source-index;
FI;

IF (PE = 1) AND ((VM = 1) OR (CPL > IOPL))
THEN (* Virtual 8086 mode, or protected mode with CPL > IOPL *)
  IF NOT I-O-Permission (DEST, width(DEST))
  THEN #GP(0);
  FI;
FI;
IF byte type of instruction
THEN
  [DX] ` [source-index]; (* Write byte at DX I/O address *)
  IF DF = 0 THEN IncDec ` 1 ELSE IncDec ` -1; FI;
  FI;
  IF OperandSize = 16
  THEN
    [DX] ` [source-index]; (* Write word at DX I/O address *)
    IF DF = 0 THEN IncDec ` 2 ELSE IncDec ` -2; FI;
    FI;
    IF OperandSize = 32
    THEN
      [DX] ` [source-index]; (* Write dword at DX I/O address *)
      IF DF = 0 THEN IncDec ` 4 ELSE IncDec ` -4; FI;
      FI;
    FI;
    source-index ` source-index + IncDec;

```

-----

## Protected Mode Exceptions

#GP(0) if the current privilege level is greater than the I/O privilege level and any of the corresponding I/O permission bits in TSS equals 1; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

-----

## Virtual 8086 Mode Exceptions

#GP(0) fault if any of the corresponding I/O permission bits in TSS equals 1; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Notes

After the OUT or OUTS instructions are run, the Pentium processor ensures that the EWBE# has been sampled active before beginning to run the next instruction. Note that the instruction may be prefetched if EWBE# is not active, but it will not run until EWBE# is sampled active.

-----

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

# POP-Pop a Word from the Stack

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
58+rw	POP <a href="#">r16</a>	X	X	X	X	X	X	Pop top of stack into word register
58+rd	POP <a href="#">r32</a>				X	X	X	Pop top of stack into dword register
8F /0	POP <a href="#">m16</a>	X	X	X	X	X	X	Pop top of stack into memory word
8F /0	POP <a href="#">m32</a>				X	X	X	Pop top of stack into memory dword
1F	POP DS	X	X	X	X	X	X	Pop top of stack into DS
07	POP ES	X	X	X	X	X	X	Pop top of stack into ES
0F A1	POP FS				X	X	X	Pop top of stack into FS
0F A9	POP GS				X	X	X	Pop top of stack into GS
17	POP SS	X	X	X	X	X	X	Pop top of stack into SS

## Description

The POP instruction replaces the previous contents of the memory, the register, or the segment register operand with the word on the top of the processor stack, addressed by SS:SP (address-size attribute of 16 bits) or SS:ESP (address-size attribute of 32 bits). The stack pointer SP is incremented by 2 for an operand-size of 16 bits or by 4 for an operand-size of 32 bits. It then points to the new top of stack.

The POP CS instruction is not a processor instruction. Popping from the stack into the CS register is accomplished with a RET instruction.

If the destination operand is a segment register (DS, ES, FS, GS, or SS), the value popped must be a selector. In protected mode, loading the selector initiates automatic loading of the descriptor information associated with that selector into the hidden part of the segment register; loading also initiates validation of both the selector and the descriptor information.

A null value (0000-0003) may be popped into the DS, ES, FS, or GS register without causing a protection exception. An attempt to reference a

segment whose corresponding segment register is loaded with a null value causes a #GP(0) exception. No memory reference occurs. The saved value of the segment register is null.

A POP SS instruction inhibits all interrupts, including NMI, until after processing of the next instruction. This allows sequential processing of POP SS and MOV ESP, EBP instructions without danger of having an invalid stack during an interrupt. However, use of the LSS instruction is the preferred method of loading the SS and ESP registers.

A POP-to-memory instruction, which uses the stack pointer (ESP) as a base register, references memory after the POP. The base used is the value of the ESP after the instruction runs.

Loading a segment register while in protected mode results in special checks and actions, as described in the following listing:

IF SS is loaded:

```
IF selector is null THEN #GP(0);
Selector index must be within its descriptor table limits ELSE
    #GP(selector);
Selector's RPL must equal CPL ELSE #GP(selector);
AR byte must indicate a writable data segment ELSE #GP(selector);
DPL in the AR byte must equal CPL ELSE #GP(selector);
Segment must be marked present ELSE #SS(selector);
Load SS register with selector;
Load SS register with descriptor;
```

IF DS, ES, FS or GS is loaded with non-null selector:

```
AR byte must indicate data or readable code segment ELSE
    #GP(selector);
IF data or nonconforming code
THEN both the RPL and the CPL must be less than or equal to DPL in
AR byte
ELSE #GP(selector);
FI;
Segment must be marked present ELSE #NP(selector);
Load segment register with selector;
Load segment register with descriptor;
```

IF DS, ES, FS, or GS is loaded with a null selector:

```
Load segment register with selector
Clear valid bit in invisible portion of register
```

-----

## Operation

```
IF StackAddrSize = 16
THEN
    IF OperandSize = 16
    THEN
        DEST ` (SS:SP); (* copy a word *)
        SP ` SP + 2;
    ELSE (* OperandSize = 32 *)
        DEST ` (SS:SP); (* copy a dword *)
        SP ` SP + 4;
    FI;
ELSE (* StackAddrSize = 32 *)
    IF OperandSize = 16
    THEN
        DEST ` (SS:ESP); (* copy a word *)
        ESP ` ESP + 2;
    ELSE (* OperandSize = 32 *)
        DEST ` (SS:ESP); (* copy a dword *)
        ESP ` ESP + 4;
    FI;
FI;
```

-----

## Protected Mode Exceptions

#GP, #SS, and #NP if a segment register is being loaded, #SS(0) if the current top of stack is not within the stack segment; #GP(0) if the

result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in real-address mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

Back-to-back PUSH/POP instruction sequences are allowed without incurring an additional clock.

The stack segment descriptor's B bit will determine the size of Stack Addr Size.

Pop ESP instructions increments the stack pointer (ESP) before data at the old top of stack is written into the destination.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## POPA/POPAD-Pop all General Registers

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
61	POPA	X	X	X	X	X	X	Pop DI, SI, BP, BX, DX, CX, and AX
61	POPAD	X	X	X	X	X	X	Pop EDI, ESI, EBP, EBX, EDX, ECX, and EAX

## Description

The POPA instruction pops the eight 16-bit general registers. However, the SP value is discarded instead of loaded into the SP register. The POPA instruction reverses a previous PUSH instruction, restoring the general registers to their values before the PUSH instruction was run. The first register popped is the DI register.

The POPAD instruction pops the eight 32-bit general registers. The ESP value is discarded instead of loaded into the ESP register. The POPAD instruction reverses the previous PUSHAD instruction, restoring the general registers to their values before the PUSHAD instruction was run. The first register popped is the EDI register.

## Operation

```
IF OperandSize = 16 (* instruction = POPA *)
THEN
  DI ` Pop();
  SI ` Pop();
  BP ` Pop();
  increment SP by 2 (* skip next 2 bytes of stack *)
  BX ` Pop();
  DX ` Pop();
  CX ` Pop();
  AX ` Pop();
ELSE (* OperandSize = 32, instruction = POPAD *)
  EDI ` Pop();
  ESI ` Pop();
  EBP ` Pop();
  increment SP by 4 (* skip next 4 bytes of stack *)
  EBX ` Pop();
  EDX ` Pop();
  ECX ` Pop();
  EAX ` Pop();
FI;
```

## Protected Mode Exceptions

#SS(0) if the starting or ending stack address is not within the stack segment; #PF(fault-code) for a page fault.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

# Virtual 8086 Mode Exceptions

Same exceptions as in real-address mode; #PF(fault-code) for a page fault.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## POPF/POPFD-Pop Stack into FLAGS or EFLAGS Register

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
9D	POPF	X	X	X	X	X	X	Pop top of stack into FLAGS
9D	POPFD			X	X	X		Pop top of stack into EFLAGS

---

## Description

The POPF and POPFD instructions pop the word or doubleword on the top of the stack and store the value in the FLAGS register. If the operand-size attribute of the instruction is 16 bits, then a word is popped and the value is stored in the FLAGS register. If the operand-size attribute is 32 bits, then a doubleword is popped and the value is stored in the EFLAGS register.

When the IOPL is less than 3 in virtual-8086 mode, the POPF instruction causes a general protection exception. When the IOPL is equal to 3 while running in virtual-8086 mode, POPF pops a word into the FLAGS register.

Refer to the Intel documentation for information about the FLAGS and EFLAGS registers. Note that bits 16 and 17 of the EFLAGS register, called the VM and RF flags, respectively, are not affected by the POPF or POPFD instruction.

The I/O privilege level is altered only when running at privilege level 0. The interrupt flag is altered only when running at a level at least as privileged as the I/O privilege level. (Real-address mode is equivalent to privilege level 0.) If a POPF instruction is run with insufficient privilege, an exception does not occur, but the privileged bits do not change.

---

# Operation

```
IF VM=0 (* Not in Virtual-8086 Mode *)
THEN
  IF OperandSize=32;
  THEN EFLAGS ` Pop() AND 277FD7H;
  ELSE FLAGE ` Pop();
  FI;
ELSE (* In Virtual-8086 Mode *)
  IF IOPL=3
  THEN
    IF OperandSize=32
    THEN
      TempEflags ` Pop();
      EFLAGS ` ((EFLAGS AND 1B3000H) OR (TempEflags AND ~ 1B3000H))
      (* VM, RF, IOPL, VIP, and VIF of EFLAGS bits are
      not modified by POPFD *)
    ELSE
      FLAGS ` Pop()
    FI;
  ELSE
    #GP(0); (* trap to virtual-8086 monitor *)
  FI;
FI;
```

---

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
*	*	*	*	*	*	*	*

All flags except the VM, RF, IOPL, VIF and VIP flags.

---

## Protected Mode Exceptions

#SS(0) if the top of stack is not within the stack segment.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

#GP(0) fault if the I/O privilege level is less than 3 in order to permit emulation. #GP(0) if an attempt is made to run POPF with an

operand-size override prefix.

# Related Information

- [Description](#)
- [Flags Affected](#)
- [Operation](#)
- [Protected Mode Exceptions](#)
- [Real Address Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

# PUSH-Push Operand onto the Stack

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
50+rw	PUSH <a href="#">r16</a>	X	X	X	X	X	X	Push register word
50+rd	PUSH <a href="#">r32</a>				X	X	X	Push register dword
FF /6	PUSH <a href="#">m16</a>	X	X	X	X	X	X	Push memory word
FF /6	PUSH <a href="#">m32</a>				X	X	X	Push memory dword
6A	PUSH <a href="#">imm8</a>		X	X	X	X	X	Push immediate byte
68	PUSH <a href="#">imm16</a>		X	X	X	X	X	Push immediate word
68	PUSH <a href="#">imm32</a>				X	X	X	Push immediate dword
0E	PUSH CS	X	X	X	X	X	X	Push CS
1E	PUSH DS	X	X	X	X	X	X	Push DS
06	PUSH ES	X	X	X	X	X	X	Push ES
0F A0	PUSH FS				X	X	X	Push FS
0F A8	PUSH GS				X	X	X	Push GS
16	PUSH SS	X	X	X	X	X	X	Push SS

## Description

The PUSH instruction decrements the stack pointer by 2 if the operand-size attribute of the instruction is 16 bits; otherwise, it decrements the stack pointer by 4. The PUSH instruction then places the operand on the new top of stack, which is pointed to by the stack pointer.

The PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction. This differs from the 8086, where the PUSH SP instruction pushes the new value (decremented by 2).

Likewise, a PUSH-from-memory instruction, which uses the stack pointer (ESP) as a base register, references memory before the PUSH. The base used is the value of the ESP before the instruction runs.

---

## Operation

```
IF StackAddrSize = 16
THEN
  IF OperandSize = 16 THEN
    SP ` SP - 2;
    (SS:SP) ` (SOURCE); (* word assignment *)
  ELSE
    SP ` SP - 4;
    (SS:SP) ` (SOURCE); (* dword assignment *)
  FI;
ELSE (* StackAddrSize = 32 *)
  IF OperandSize = 16
  THEN
    ESP ` ESP - 2;
    (SS:ESP) ` (SOURCE); (* word assignment *)
  ELSE
    ESP ` ESP - 4;
    (SS:ESP) ` (SOURCE); (* dword assignment *)
  FI;
FI;
```

---

## Protected Mode Exceptions

#SS(0) if the new value of the SP or ESP register is outside the stack segment limit; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

None; if the SP or ESP register is 1, the processor shuts down due to a lack of stack space.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in real-address mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

When used with an operand in memory, the PUSH instruction takes longer to run than a two-instruction sequence, which moves the operand through a register.

Back-to-back PUSH/POP instruction sequences are allowed without incurring an additional clock.

Selective pushes write only the top of the stack.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## PUSHA/PUSHAD-Push all General Registers

---

### Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
60	PUSHA	X	X	X	X	X	X	Push AX, CX, DX, BX, SP, BP, SI, and DI
60	PUSHAD	X	X	X	X	X	X	Push EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI

---

### Description

The PUSHA and PUSHAD instructions save the 16-bit or 32-bit general registers, respectively, on the processor stack. The PUSHA instruction decrements the stack pointer (SP) by 16 to hold the eight word values. The PUSHAD instruction decrements the stack pointer (ESP) by 32 to hold the eight doubleword values. Because the registers are pushed onto the stack in the order in which they were given, they appear in the 16 or 32 new stack bytes in reverse order. The last register pushed is the DI or EDI register.

---

### Operation

```
IF OperandSize = 16 (* PUSHAD instruction *)
THEN Temp` (SP);
  Push(AX);
  Push(CX);
  Push(DX);
  Push(BX);
  Push(Temp);
  Push(BP);
  Push(SI);
  Push(DI);
ELSE (* OperandSize = 32, PUSHAD instruction *)
  Temp` (ESP);
  Push(EAX);
  Push(ECX);
  Push(EDX);
  Push(EBX);
  Push(Temp);
  Push(ESP);
  Push(ESI);
  Push(EDI);
FI;
```

---

## Protected Mode Exceptions

#SS(0) if the starting or ending stack address is outside the stack segment limit; #PF(fault-code) for a page fault.

---

## Real Address Mode Exceptions

Before running the PUSHAD or PUSHAD instruction, the Pentium processor shuts down if the SP or ESP register equals 1, 3, or 5; if the SP or ESP register equals 7, 9, 11, 13, or 15, exception 13 occurs.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in real-address mode; #PF(fault-code) for a page fault.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

# PUSHF/PUSHFD-Push Flags Register onto the Stack

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
9C	PUSHF	X	X	X	X	X	X	Push FLAGS
9C	PUSHFD			X	X	X		Push EFLAGS

## Description

The PUSHF instruction decrements the stack pointer by 2 and copies the FLAGS register to the new top of stack; the PUSHFD instruction decrements the stack pointer by 4, and the EFLAGS register is copied to the new top of stack, which is pointed to by SS:ESP. Refer to the Intel documentation for information on the EFLAGS register.

## Operation

```
IF VM=0 (* Not in Virtual-8086 Mode *)
THEN
  IF OperandSize = 32
  THEN push(EFLAGS AND 0FCFFFFH); (* VM and RF EFLAG bits are cleared *)
  ELSE push(FLAGS);
  FI;
ELSE (* In Virtual-8086 Mode *)
  IF IOPL=3
  THEN
    IF OperandSize = 32
    THEN push(EFLAGS AND 0FCFFFFH); (* VM and RF EFLAGS bits are cleared *)
    ELSE push(FLAGS);
    FI;
  ELSE
    #GP(0); (* Trap to virtual-8086 monitor *)
  FI;
FI;
```

## Protected Mode Exceptions

#SS(0) if the new value of the ESP register is outside the stack segment boundaries.

## Real Address Mode Exceptions

None; the processor shuts down due to a lack of stack space.

# Virtual 8086 Mode Exceptions

#GP(0) fault if the I/O privilege level is less than 3, to permit emulation.

## Related Information

- Description
- Flags Affected
- Operation
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

## RCL/RCR/ROL/ROR-Rotate

## Details Table

Encoding	Instruction	0 1 2 3 4 5	Description
D0 /2	RCL <code>r/m8,1</code>	X X X X X X	Rotate 9 bits (CF, r/m byte) left once
D2 /2	RCL <code>r/m8,CL</code>	X X X X X X	Rotate 9 bits (CF, r/m byte) left CL times
C0 /2 ib	RCL <code>r/m8,imm8</code>	X X X X X X	Rotate 9 bits (CF, r/m byte) left imm8 times
D1 /2	RCL <code>r/m16,1</code>	X X X X X X	Rotate 17 bits (CF, r/m word) left once
D3 /2	RCL <code>r/m16,CL</code>	X X X X X X	Rotate 17 bits (CF, r/m word) left CL times
C1 /2 ib	RCL <code>r/m16,imm8</code>	X X X X X X	Rotate 17 bits (CF, r/m word) left imm8 times
D1 /2	RCL <code>r/m32,1</code>	X X X	Rotate 33 bits (CF, r/m dword) left once
D3 /2	RCL <code>r/m32,CL</code>	X X X	Rotate 33 bits (CF, r/m dword) left CL times

C1 /2 ib	RCL <a href="#">r/m32,imm8</a>	X X X	Rotate 33 bits (CF, r/m dword) left imm8 times
D0 /3	RCR <a href="#">r/m8,1</a>	X X X X X X	Rotate 9 bits (CF, r/m byte) right once
D2 /3	RCR <a href="#">r/m8,CL</a>	X X X X X X	Rotate 9 bits (CF, r/m byte) right CL times
C0 /3 ib	RCR <a href="#">r/m8,imm8</a>	X X X X X	Rotate 9 bits (CF, r/m byte) right imm8 times
D1 /3	RCR <a href="#">r/m16,1</a>	X X X X X X	Rotate 17 bits (CF, r/m word) right once
D3 /3	RCR <a href="#">r/m16,CL</a>	X X X X X X	Rotate 17 bits (CF, r/m word) right CL times
C1 /3 ib	RCR <a href="#">r/m16,imm8</a>	X X X X X	Rotate 17 bits (CF, r/m word) right imm8 times
D1 /3	RCR <a href="#">r/m32,1</a>	X X X	Rotate 33 bits (CF, r/m dword) right once
D3 /3	RCR <a href="#">r/m32,CL</a>	X X X	Rotate 33 bits (CF, r/m dword) right CL times
C1 /3 ib	RCR <a href="#">r/m32,imm8</a>	X X X	Rotate 33 bits (CF, r/m dword) right imm8 times
D0 /0	ROL <a href="#">r/m8,1</a>	X X X X X X	Rotate 8 bits (r/m byte) left once
D2 /0	ROL <a href="#">r/m8,CL</a>	X X X X X X	Rotate 8 bits (r/m byte) left CL times
C0 /0 ib	ROL <a href="#">r/m8,imm8</a>	X X X X X	Rotate 8 bits (r/m byte) left imm8 times
D1 /0	ROL <a href="#">r/m16,1</a>	X X X X X X	Rotate 16 bits (r/m word) left once
D3 /0	ROL <a href="#">r/m16,CL</a>	X X X X X X	Rotate 16 bits (r/m word) left CL times
C1 /0 ib	ROL <a href="#">r/m16,imm8</a>	X X X X X	Rotate 16 bits (r/m word) left imm8 times
D1 /0	ROL <a href="#">r/m32,1</a>	X X X	Rotate 32 bits (r/m dword) left once
D3 /0	ROL <a href="#">r/m32,CL</a>	X X X	Rotate 32 bits (r/m dword) left CL times
C1 /0 ib	ROL <a href="#">r/m32,imm8</a>	X X X	Rotate 32 bits (r/m dword) left imm8 times
D0 /1	ROR <a href="#">r/m8,1</a>	X X X X X X	Rotate 8 bits (r/m byte) right once
D2 /1	ROR <a href="#">r/m8,CL</a>	X X X X X X	Rotate 8 bits (r/m byte) right CL times
C0 /1 ib	ROR <a href="#">r/m8,imm8</a>	X X X X X	Rotate 8 bits (r/m byte) right imm8 times
D1 /1	ROR <a href="#">r/m16,1</a>	X X X X X X	Rotate 16 bits (r/m word) right once
D3 /1	ROR <a href="#">r/m16,CL</a>	X X X X X X	Rotate 16 bits (r/m word) right CL times
C1 /1 ib	ROR <a href="#">r/m16,imm8</a>	X X X X X	Rotate 16 bits (r/m word) right imm8 times
D1 /1	ROR <a href="#">r/m32,1</a>	X X X	Rotate 32 bits (r/m dword) right once
D3 /1	ROR <a href="#">r/m32,CL</a>	X X X	Rotate 32 bits (r/m dword) right CL times
C1 /1 ib	ROR <a href="#">r/m32,imm8</a>	X X X	Rotate 32 bits (r/m dword) right imm8 times

---

## Description

Each rotate instruction shifts the bits of the register or memory operand given. The left rotate instructions shift all the bits upward, except for the top bit, which is returned to the bottom. The right rotate instructions do the reverse: the bits shift downward until the bottom bit arrives at the top.

For the RCL and RCR instructions, the CF flag is part of the rotated quantity. The RCL instruction shifts the CF flag into the bottom bit and shifts the top bit into the CF flag; the RCR instruction shifts the CF flag into the top bit and shifts the bottom bit into the CF flag. For the ROL and ROR instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other.

The rotate is repeated the number of times indicated by the second operand, which is either an immediate number or the contents of the CL register. To reduce the maximum instruction processing time, the Pentium processor does not allow rotation counts greater than 31. If a rotation count greater than 31 is attempted, only the bottom five bits of the rotation are used. The 8086 does not mask rotation counts. The Pentium processor in Virtual 8086 Mode does mask rotation counts.

The OF flag is defined only for the single-rotate forms of the instructions (second operand is a 1). It is undefined in all other cases. For left shifts/rotates, the CF bit after the shift is XORed with the high-order result bit. For right shifts/rotates, the high-order two bits of the result are XORed to get the OF flag.

---

## Operation

```
(* ROL - Rotate Left *)
temp ` COUNT;
WHILE (temp <> 0)
DO
    tmpcf ` high-order bit of (r/m);
    r/m ` r/m * 2 + (tmpcf);
    temp ` temp - 1;
OD;
IF COUNT = 1
THEN
    IF high-order bit of r/m <> CF
    THEN OF ` 1;
    ELSE OF ` 0;
    FI;
ELSE OF ` undefined;
FI;
(* ROR - Rotate Right *)
temp ` COUNT;
WHILE (temp <> 0)
DO
    tmpcf ` low-order bit of (r/m);
    r/m ` r/m / 2 + (tmpcf * 2widty(r/m));
    temp ` temp - 1;
OD;
IF COUNT = 1
THEN
    IF (high-order bit of r/m) <> (bit next to high-order bit of r/m)
    THEN OF ` 1;
    ELSE OF ` 0;
    FI;
ELSE OF ` undefined;
FI;
```

---

## Flags Affected

OF DF IF SF ZF AF PF CF

\* \*

The OF flag is affected only for single-bit rotates; the OF flag is undefined for multi-bit rotates; the CF flag contains the value of the bit shifted into it; the SF, ZF, AF, and PF flags are not affected.

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

## RDMSR-Read from Model Specific Register

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F 32	RDMSR							X Read Model Specific Register indicated by ECX into EDX:EAX

## Description

The value in ECX specifies one of the 64-bit Model Specific Registers of the Pentium processor. The content of that Model-Specific Register is copied into EDX:EAX. EDX is loaded with the high-order 32 bits, and EAX is loaded with the low-order 32 bits.

The following values are used to select model specific registers on the Pentium processor.

VALUE	REGISTER NAME	DESCRIPTION
00H	Machine Check Address	Stores address of cycle causing the exception
01H	Machine Check Type	Stores type of cycle causing the exception

For other values used to perform cache, TLB, and BTB testing and performance monitoring, see the Intel documentation.

## Operation

EDX:EAX ← MSR[ECX];

## Protected Mode Exceptions

#GP(0) if either the current privilege level is not 0 or the value in ECX does not specify a Model-Specific Register that is implemented in the Pentium processor.

## Real Address Mode Exceptions

#GP if the value in ECX does not specify a Model-Specific Register that is implemented in the Pentium processor.

## Virtual 8086 Mode Exceptions

#GP(0) if instruction processing is attempted.

# Notes

This instruction must be run at privilege level 0 or in real-address mode; otherwise, a protection exception will be generated.

If less than 64 bits are implemented in a model specific register, the value returned to EDX:EAX, in the locations corresponding to the unimplemented bits, is unpredictable.

RDMSR is used to read the content of Model-Specific Registers that control functions for testability, execution tracing, performance monitoring and machine check errors. Refer to the *Pentium(TM) Processor Data Book* for more information.

The values 3H, 0FH, and values above 13H are reserved. Do not run RDMSR with reserved values in ECX.

## Related Information

- [Description](#)
- [Flags Affected](#)
- [Notes](#)
- [Operation](#)
- [Protected Mode Exceptions](#)
- [Real Address Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

## RDTSC-Read Time Stamp Counter

### Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F 31	RDTSC							<span>X</span> Read Time Stamp Counter into EDX:EAX

### Description

# Operation

-----

## Protected Mode Exceptions

-----

## Real Address Mode Exceptions

-----

## Virtual 8086 Mode Exceptions

-----

## Related Information

- [Description](#)
- [Flags Affected](#)
- [Operation](#)
- [Protected Mode Exceptions](#)
- [Real Address Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

-----

## REP/REPE/REPZ/REPNE/REPNZ-Repeat Following String Operati

-----

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
----------	-------------	---	---	---	---	---	---	-------------

F3 6C	REP INS <i>r/m8</i> ,DX	X X X X X	Input (E)CX bytes from port DX into ES:[(E)DI]
F3 6D	REP INS <i>r/m16</i> ,DX	X X X X X	Input (E)CX words from port DX into ES:[(E)DI]
F3 6D	REP INS <i>r/m32</i> ,DX	X X X	Input (E)CX dwords from port DX into ES:[(E)DI]
F3 AC	REP LODS AL	X X X X X X	Load (E)CX bytes from [(E)SI] to EDX
F3 AD	REP LODS AX	X X X X X X	Load (E)CX words from [(E)SI] to EDX
F3 AD	REP LODS EAX	X X X	Load (E)CX dwords from [(E)SI] to EDX
F3 A4	REP MOVS <i>m8</i> , <i>m8</i>	X X X X X X	Move (E)CX bytes from [(E)SI] to ES:[(E)DI]
F3 A5	REP MOVS <i>m16</i> , <i>m16</i>	X X X X X X	Move (E)CX words from [(E)SI] to ES:[(E)DI]
F3 A5	REP MOVS <i>m32</i> , <i>m32</i>	X X X	Move (E)CX dwords from [(E)SI] to ES:[(E)DI]
F3 6E	REP OUTS DX, <i>r/m8</i>	X X X X X	Output (E)CX bytes from [(E)SI] to port DX
F3 6F	REP OUTS DX, <i>r/m16</i>	X X X X X	Output (E)CX words from [(E)SI] to port DX
F3 6F	REP OUTS DX, <i>r/m32</i>	X X X	Output (E)CX dwords from [(E)SI] to port DX
F3 AA	REP STOS <i>m8</i>	X X X X X X	Store (E)CX bytes at ES:[(E)DI] from AL
F3 AB	REP STOS <i>m16</i>	X X X X X X	Store (E)CX words at ES:[(E)DI] from AX
F3 AB	REP STOS <i>m32</i>	X X X	Store (E)CX dwords at ES:[(E)DI] from EAX
F3 A6	REPE CMPS <i>m8</i> , <i>m8</i>	X X X X X X	Find nonmatching bytes in ES:[(E)DI] and [(E)SI]
F3 A7	REPE CMPS <i>m16</i> , <i>m16</i>	X X X X X X	Find nonmatching words in ES:[(E)DI] and [(E)SI]
F3 A7	REPE CMPS <i>m32</i> , <i>m32</i>	X X X	Find nonmatching dwords in ES:[(E)DI] and [(E)SI]
F3 AE	REPE SCAS <i>m8</i> , <i>m8</i>	X X X X X X	Find non-AL byte starting at ES:[(E)DI]
F3 AF	REPE SCAS <i>m16</i> , <i>m16</i>	X X X X X X	Find non-AX word starting at ES:[(E)DI]
F3 AF	REPE SCAS <i>m32</i> , <i>m32</i>	X X X	Find non-EAX dword starting at ES:[(E)DI]
F2 A6	REPNE CMPS <i>m8</i> , <i>m8</i>	X X X X X X	Find matching bytes in ES:[(E)DI] and [(E)SI]
F2 A7	REPNE CMPS <i>m16</i> , <i>m16</i>	X X X X X X	Find matching words in ES:[(E)DI] and [(E)SI]
F2 A7	REPNE CMPS <i>m32</i> , <i>m32</i>	X X X	Find matching dwords in ES:[(E)DI] and [(E)SI]
F2 AE	REPNE SCAS <i>m8</i> , <i>m8</i>	X X X X X X	Find AL, starting at ES:[(E)DI]
F2 AF	REPNE SCAS <i>m16</i> , <i>m16</i>	X X X X X X	Find AX, starting at ES:[(E)DI]
F2 AF	REPNE SCAS <i>m32</i> , <i>m32</i>	X X X	Find EAX, starting at ES:[(E)DI]

-----

# Description

The REP, REPE (repeat while equal), and REPNE (repeat while not equal) prefixes are applied to string operation. Each prefix causes the string instruction that follows to be repeated the number of times indicated in the count register or (for the REPE and REPNE prefixes) until the indicated condition in the ZF flag is no longer met.

Synonymous forms of the REPE and REPNE prefixes are the REPZ and REPNZ prefixes, respectively.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct.

The precise action for each iteration is as follows:

1. If the address-size attribute is 16 bits, use the CX register for the count register; if the address-size attribute is 32 bits, use the ECX register for the count register.
2. Check the count register. If it is zero, exit the iteration, and move to the next instruction.
3. Acknowledge any pending interrupts.
4. Perform the string operation once.
5. Decrement the CX or count register by one; no flags are modified.
6. Check the ZF flag if the string operation is a SCAS or CMPS instruction. If the repeat condition does not hold, exit the iteration and move to the next instruction. Exit the iteration if the prefix is REPE and the ZF flag is 0 (the last comparison was not equal), or if the prefix is REPNE and the ZF flag is one (the last comparison was equal).
7. Return to step 2 for the next iteration.

Repeated CMPS and SCAS instructions can be exited if the count is exhausted or if the ZF flag fails the repeat condition. These two cases can be distinguished by using either the JCXZ instruction, or by using the conditional jumps that test the ZF flag (the JZ, JNZ, and JNE instructions).

---

## Operation

```
IF AddressSize = 16
THEN use CX for CountReg;
ELSE (* AddressSize = 32 *) use ECX for CountReg;
FI;
WHILE CountReg <> 0
DO
  service pending interrupts (if any);
  perform primitive string instruction;
  CountReg = CountReg - 1;
  IF primitive operation is CMPSB, CMPSW, CMPSD, SCASB,
  SCASW, or SCASD
  THEN
    IF (instruction is REP/REPE/REPZ) AND (ZF=0)
    THEN exit WHILE loop
  ELSE
    IF (instruction is REPNZ or REPNE) AND (ZF=1)
    THEN exit WHILE loop;
  FI;
FI;
FO;
OD;
```

---

## Flags Affected

OF DF IF SF ZF AF PF CF

\*

The ZF flag is affected by the REP CMPS and REP SCAS as described above.

-----

## Notes

- Not all I/O ports can handle the rate at which the REP INS and REP OUTS instructions run.
  - Do not use the REP prefix with the LOOP instruction. Proper LOOP operation is not guaranteed when used with the REP prefix and the effect of this combination is unpredictable.
  - The behavior of the REP prefix is undefined when used with non-string instructions.
  - When a page fault occurs during CMPS or SCAS instructions that are prefixed with REPNE, the EFLAGS value is restored to the state prior to the processing of the instruction. Because SCAS and CMPS do not use EFLAGS as an input, the processor can resume the instruction after the page fault handler.
- 

## Related Information

- [Description](#)
  - [Flags Affected](#)
  - [Notes](#)
  - [Operation](#)
  - [Protected Mode Exceptions](#)
  - [Protected Mode Exceptions](#)
  - [Virtual 8086 Mode Exceptions](#)
- 

## RET-Return from Procedure

-----

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
C3	RET/RETN	X	X	X	X	X	X	Return (near) to caller
CB	RET/RETF	X	X	X	X	X	X	Return (far) to caller, same privilege
CB	RET/RETF		X	X	X	X		Return (far), lesser privilege,

switch stacks			
C2 iw	RET/RETN <i>imm16</i>	X X X X X X	Return (near), pop <i>imm16</i> bytes
CA iw	RET/RETF <i>imm16</i>	X X X X X X	Return (far), same privilege, pop <i>imm16</i> bytes
CA iw	RET/RETF <i>imm16</i>	X X X X	Return (far), lesser privilege, pop <i>imm16</i> bytes

## Description

The RET instruction transfers control to a return address located on the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional numeric parameter to the RET instruction gives the number of stack bytes (OperandMode=16) or words (OperandMode=32) to be released after the return address is popped. These items are typically used as input parameters to the procedure called.

For the intrasegment (near) return, the address on the stack is a segment offset, which is popped into the instruction pointer. The CS register is unchanged. For the intersegment (far) return, the address on the stack is a long pointer. The offset is popped first, followed by the selector.

In real mode, the CS and IP registers are loaded directly. In Protected Mode, an intersegment return causes the processor to check the descriptor addressed by the return selector. The AR byte of the descriptor must indicate a code segment of equal or lesser privilege (or greater or equal numeric value) than the current privilege level. Returns to a lesser privilege level cause the stack to be reloaded from the value saved beyond the parameter block.

The DS, ES, FS, and GS segment registers can be cleared by the RET instruction during an interlevel transfer. If there registers refer to segments that cannot be used by the new privilege level, they are cleared to prevent unauthorized access from the new privilege level.

## Operation

```

IF instruction = near RET
THEN;
  IF OperandSize = 16
  THEN
    IP ` Pop();
    EIP ` EIP AND 0000FFFFH;
  ELSE (* OperandSize = 32 *)
    EIP ` Pop();
  FI;
  IF instruction has immediate operand THEN eSP ` eSP + imm16; FI;
FI;

IF (PE = 0 OR (PE = 1 AND VM = 1))
(* real mode or virtual 8086 mode *)
AND instruction = far RET
THEN;
  IF OperandSize = 16
  THEN
    IP ` Pop();
    EIP ` EIP AND 0000FFFFH;
    CS ` Pop(); (* 16-bit pop *)
  ELSE (* OperandSize = 32 *)
    EIP ` Pop ();
    CS ` Pop(); (* 32-bit pop, high-order 16-bits discarded *)
  FI;
  IF instruction has immediate operand THEN eSP ` eSP + imm16; FI;
FI;

IF (PE = 1 AND VM = 0) (* Protected mode, not V86 mode *)
AND instruction = far RET

```

```

THEN
  IF OperandSize=32
  THEN Third word on stack must be within stack limits else #SS(0);
  ELSE Second word on stack must be within stack limits else #SS(0);
  FI;
  Return selector RPL must be  $\frac{3}{4}$  CPL ELSE #GP(return selector)
  IF return selector RPL = CPL
  THEN GOTO SAME-LEVEL;
  ELSE GOTO OUTER-PRIVILEGE-LEVEL;
  FI;
FI;

```

#### SAME-LEVEL:

```

  Return selector must be non-null ELSE #GP(0)
  Selector index must be within its descriptor table limits ELSE
    #GP(selector)
  Descriptor AR byte must indicate code segment ELSE #GP(selector)
  IF non-conforming
  THEN code segment DPL must equal CPL;
  ELSE #GP(selector);
  FI;
  IF conforming
  THEN code segment DPL must be  $\frac{3}{4}$  CPL;
  ELSE #GP(selector);
  FI;
  Code segment must be present ELSE #NP(selector);
  Top word on stack must be within stack limits ELSE #SS(0);
  IP must be in code segment limit ELSE #GP(0);
  IF OperandSize=32
  THEN
    Load CS:EIP from stack
    Load CS register with descriptor
    Increment eSP by 8 plus the immediate offset if it exists
  ELSE (* OperandSize=16 *)
    Load CS:IP from stack
    Load CS register with descriptor
    Increment eSP by 4 plus the immediate offset if it exists
  FI;

```

#### OUTER-PRIVILEGE-LEVEL:

```

  IF OperandSize=32
  THEN Top (16+immediate) bytes on stack must be within stack limits
    ELSE #SS(0);
  ELSE Top (8+immediate) bytes on stack must be within stack limits ELSE
    #SS(0);
  FI;
  Examine return CS selector and associated descriptor:
    Selector must be non-null ELSE #GP(0);
    Selector index must be within its descriptor table limits ELSE
      #GP(selector)
    Descriptor AR byte must indicate code segment ELSE #GP(selector);
    IF non-conforming
    THEN code segment DPL must equal return selector RPL
    ELSE #GP(selector);
    FI;

    IF conforming
    THEN code segment DPL must be  $\frac{3}{4}$  return selector RPL;
    ELSE #GP(selector);
    FI;
    Segment must be present ELSE #NP(selector)
  Examine return SS selector and associated descriptor:
    Selector must be non-null ELSE #GP(0);
    Selector index must be within its descriptor table limits
      ELSE #GP(selector);
    Selector RPL must equal the RPL of the return CS selector ELSE
      #GP(selector);
    Descriptor AR byte must indicate a writable data segment ELSE
      #GP(selector);
    Descriptor DPL must equal the RPL of the return CS selector ELSE
      #GP(selector);
    Segment must be present ELSE #NP(selector);
  IP must be in code segment limit ELSE #GP(0);
  Set CPL to the RPL of the return CS selector;
  IF OperandSize=32
  THEN

```

```

    Load CS:IP from stack;
    Set CS RPL to CPL;
    Increment eSP by 8 plus the immediate offset if it exists;
    Load SS:eSP from stack;
ELSE (* OperandSize=16 *)
    Load CS:IP from stack;
    Set CS RPL to CPL;
    Increment eSP by 4 plus the immediate offset if it exists;
    Load SS:eSP from stack;
FI;
Load the CS register with the return CS descriptor;
Load the SS register with the return SS descriptor;
For each of ES, FS, GS, and DS
DO
    IF the current register setting is not valid for the outer level,
        set the register to null (selector `AR` 0);
    To be valid, the register setting must satisfy the following properties:
        Selector index must be within descriptor table limits;
        Descriptor AR byte must indicate data or readable code segment;
        IF segment is data or non-conforming code, THEN
            DPL must be CPL, or DPL must be RPL;
FI;
OD;

```

---

## Protected Mode Exceptions

#GP, #NP, or #SS, as described under "Operation" above; #PF(fault-code) for a page fault.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would be outside the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## ROL/ROR-Rotate

See entry for RCL/RCR/ROL/ROR.

# RSM-Resume from System Management Mode

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F AA	RSM							X Resume operation of interrupted program

## Description

The processor state is restored from the dump created upon entrance to SMM. Note, however, that the contents of the model-specific registers are not affected. The processor leaves SMM and returns control to the interrupted application or operating system. If the processor detects any invalid state information, it enters the shutdown state. This happens in any of the following situations:

- The value stored in the State Dump Base field is not a 32Kbyte aligned address.
- Any reserved bit of CR4 is set to 1.
- Any combination of bits in CR0 is illegal; namely, (PG=1 and PE=0) or (NW=1 and CD=0).

## Operation

Resume operation of a program interrupted by a System Management Mode interrupt.

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
*	*	*	*	*	*	*	*

All

---

## Protected Mode Exceptions

#UD if an attempt is made to run this instruction when the processor is not in System Management Mode.

---

## Real Address Mode Exceptions

#UD if an attempt is made to run this instruction when the processor is not in System Management Mode.

---

## Virtual 8086 Mode Exceptions

#UD if an attempt is made to run this instruction when the processor is not in System Management Mode.

---

## Notes

Refer to the Intel documentation for more information about System Management Mode and the behavior of the RSM instruction.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## SAHF-Store AH into Flags

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
9E	SAHF	X	X	X	X	X	X	Store AH into flags (SF, ZF, xx, AF, xx, PF, xx, CF)

## Description

The SAHF instruction loads the SF, ZF, AF, PF,A and CF flags with values from the AH register, from bits 7, 6, 4, 2, and 0, respectively.

## Operation

SF:ZF:xx:AF:xx:PF:xx:CF` AH;

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
			*	*	*	*	*

The SF, ZF, AF, PF, and CF flags are loaded with values from the AH register.

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Protected Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

## SAL/SAR/SHL/SHR-Shift Instructions

# Details Table

Encoding	Instruction	0 1 2 3 4 5	Description
D0 /4	SAL <a href="#">r/m8,1</a>	X X X X X X	Multiply r/m byte by 2, once
D2 /4	SAL <a href="#">r/m8,CL</a>	X X X X X X	Multiply r/m byte by 2, CL times
C0 /5 ib	SAL <a href="#">r/m8,imm8</a>	X X X X X	Multiply r/m byte by 2, imm8 times
D1 /4	SAL <a href="#">r/m16,1</a>	X X X X X X	Multiply r/m word by 2, once
D1 /4	SAL <a href="#">r/m32,1</a>	X X X	Multiply r/m dword by 2, once
D3 /4	SAL <a href="#">r/m16,CL</a>	X X X X X X	Multiply r/m word by 2, CL times
D3 /4	SAL <a href="#">r/m32,CL</a>	X X X	Multiply r/m dword by 2, CL times
C1 /4 ib	SAL <a href="#">r/m16,imm8</a>	X X X X X	Multiply r/m word by 2, imm8 times
C1 /4 ib	SAL <a href="#">r/m32,imm8</a>	X X X	Multiply r/m dword by 2, imm8 times
D0 /7	SAR <a href="#">r/m8,1</a>	X X X X X X	Signed divide r/m byte by 2, once
D2 /7	SAR <a href="#">r/m8,CL</a>	X X X X X X	Signed divide r/m byte by 2, CL times
C0 /7 ib	SAR <a href="#">r/m8,imm8</a>	X X X X X	Signed divide r/m byte by 2, imm8 times
D1 /7	SAR <a href="#">r/m16,1</a>	X X X X X X	Signed divide r/m word by 2, once
D1 /7	SAR <a href="#">r/m32,1</a>	X X X	Signed divide r/m dword by 2, once
D3 /7	SAR <a href="#">r/m16,CL</a>	X X X X X X	Signed divide r/m word by 2, CL times
D3 /7	SAR <a href="#">r/m32,CL</a>	X X X	Signed divide r/m dword by 2, CL times
C1 /7 ib	SAR <a href="#">r/m16,imm8</a>	X X X X X	Signed divide r/m word by 2, imm8 times
C1 /7 ib	SAR <a href="#">r/m32,imm8</a>	X X X	Signed divide r/m dword by 2, imm8 times
D0 /4	SHL <a href="#">r/m8,1</a>	X X X X X X	Multiply r/m byte by 2, once
D0 /4	SHL <a href="#">r/m8,CL</a>	X X X X X X	Multiply r/m byte by 2, CL times
C0 /4 ib	SHL <a href="#">r/m8,imm8</a>	X X X X X	Multiply r/m byte by 2, imm8 times
D1 /4	SHL <a href="#">r/m16,1</a>	X X X X X X	Multiply r/m word by 2, once
D1 /4	SHL <a href="#">r/m32,1</a>	X X X	Multiply r/m dword by 2, once
D3 /4	SHL <a href="#">r/m16,CL</a>	X X X X X X	Multiply r/m word by 2, CL times
D3 /4	SHL <a href="#">r/m32,CL</a>	X X X	Multiply r/m dword by 2, CL times
C1 /4 ib	SHL <a href="#">r/m16,imm8</a>	X X X X X	Multiply r/m word by 2, imm8 times
C1 /4 ib	SHL <a href="#">r/m32,imm8</a>	X X X	Multiply r/m dword by 2, imm8 times
D0 /5	SHR <a href="#">r/m8,1</a>	X X X X X X	Signed divide r/m byte by 2, once
D2 /5	SHR <a href="#">r/m8,CL</a>	X X X X X X	Signed divide r/m byte by 2, CL times
C0 /5 ib	SHR <a href="#">r/m8,imm8</a>	X X X X X	Signed divide r/m byte by 2, imm8 times
D1 /5	SHR <a href="#">r/m16,1</a>	X X X X X X	Signed divide r/m word by 2, once

D1	/5	SHR	<i>r/m32</i> ,1	X X X	Signed divide <i>r/m</i> dword by 2, once
D3	/5	SHR	<i>r/m16</i> ,CL	X X X X X	Signed divide <i>r/m</i> word by 2, CL times
D3	/5	SHR	<i>r/m32</i> ,CL	X X X	Signed divide <i>r/m</i> dword by 2, CL times
C1	/5 ib	SHR	<i>r/m16</i> ,imm8	X X X X X	Signed divide <i>r/m</i> word by 2, imm8 times
C1	/5 ib	SHR	<i>r/m32</i> ,imm8	X X X	Signed divide <i>r/m</i> dword by 2, imm8 times

## Description

The SAL instruction (or its synonym, SHL) shifts the bits of the operand upward. The high-order bit is shifted into the CF flag, and the low-order bit is cleared.

The SAR and SHR instructions shift the bits of the operand downward. The low-order bit is shifted into the CF flag. The effect is to divide the operand by two. The SAR instruction performs a signed divide with rounding toward negative infinity (not the same as the IDIV instruction); the high-order bit remains the same. The SHR instruction performs an unsigned divide; the high-order bit is cleared.

The shift is repeated the number of times indicated by the second operand, which is either an immediate number or the contents of the CL register. To reduce the maximum processing time, the Pentium processor does not allow shift counts greater than 31. If a shift count greater than 31 is attempted, only the bottom five bits of the shift count are used. (The 8086 uses all eight bits of the shift count.)

The OF flag is affected only if the single-shift forms of the instructions are used. For left shifts, the OF flag is cleared if the high bit of the answer is the same as the result of the CF flag (i.e., the top two bits of the original operand were the same); the OF flag is set if they are different. For the SAR instruction, the OF flag is cleared for all single shifts. For the SHR instruction, the OF flag is set for the high-order bit of the original operand.

## Operation

```
(* COUNT is the second parameter *)
(temp) ← COUNT;
WHILE (temp > 0)
DO
  IF instruction is SAL or SHL
  THEN CF ← high-order bit of r/m;
  FI;
  IF instruction is SAR or SHR
  THEN CF ← low-order bit of r/m;
  FI;
  IF instruction = SAL or SHL
  THEN r/m ← r/m * 2;
  FI;
  IF instruction = SAR
  THEN r/m ← r/m / 2 (*Signed divide, rounding toward negative infinity*);
  FI;
  IF instruction = SHR
  THEN r/m ← r/m / 2; (* Unsigned divide *);
  FI;
  temp ← temp - 1;
OD;
* Determine overflow for the various instructions *)
IF COUNT = 1
THEN
  IF instruction is SAL or SHL
  THEN OF ← high-order bit of r/m <> (CF);
  FI;
  IF instruction is SAR
```

```
THEN OF ` 0;
FI;
IF instruction is SHR
THEN OF ` high-order bit of operand;
FI;
ELSE OF ` undefined;
FI;
```

-----

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
*			*	*	?	*	*

If count = 0, the flags are not affected.

The CF flag contains the value of the last bit shifted out. The CF flag is undefined for SHL and SHR instructions in which the shift lengths are greater than or equal to the size of the operand to be shifted.

The OF flag is affected for single shifts; the OF flag is undefined for multiple shifts; the CF, ZF, PF, and SF flags are set according to the result.

-----

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

-----

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

# SBB-Integer Subtraction with Borrow

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
1C ib	SBB AL,imm8	X	X	X	X	X	X	Subtract with borrow, immediate byte from AL
1D iw	SBB AX,imm16	X	X	X	X	X	X	Subtract with borrow, immediate word from AX
1D id	SBB EAX,imm32				X	X	X	Subtract with borrow, immediate dword from EAX
80 /3 ib	SBB r/m8,imm8	X	X	X	X	X	X	Subtract with borrow, immediate byte from r/m byte
81 /3 iw	SBB r/m16,imm16	X	X	X	X	X	X	Subtract with borrow, immediate word from r/m word
81 /3 id	SBB r/m32,imm32				X	X	X	Subtract with borrow, immediate dword from r/m dword
83 /3 ib	SBB r/m16,imm8	X	X	X	X	X	X	Subtract with borrow, sign-extended immediate byte from r/m word
83 /3 ib	SBB r/m32,imm8				X	X	X	Subtract with borrow, sign-extended immediate byte from r/m dword
18 /r	SBB r/m8,r8	X	X	X	X	X	X	Subtract with borrow, byte register from r/m byte
19 /r	SBB r/m16,r16	X	X	X	X	X	X	Subtract with borrow, word register from r/m word
19 /r	SBB r/m32,r32				X	X	X	Subtract with borrow, dword register from r/m dword
1A /r	SBB r8,r/m8	X	X	X	X	X	X	Subtract with borrow, r/m byte from byte register
1B /r	SBB r16,r/m16	X	X	X	X	X	X	Subtract with borrow, r/m word from word register
1B /r	SBB r32,r/m32				X	X	X	Subtract with borrow, r/m dword from dword register

## Description

The SBB instruction adds the second operand (SRC) to the CF flag and subtracts the result from the first operand (DEST). The result of the subtraction is assigned to the first operand (DEST), and the flags are set accordingly.

When an immediate byte value is subtracted from a word operand, the immediate value is first sign-extended.

---

## Operation

F SRC is a byte and DEST is a word or dword  
THEN DEST = DEST - (SignExtend(SRC) + CF)  
ELSE DEST = DEST - (SRC + CF);

---

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
*			*	*	*	*	*

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

---

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Related Information

[Description](#)

Flags Affected

Operation

Protected Mode Exceptions

Real Address Mode Exceptions

Virtual 8086 Mode Exceptions

# SCAS/SCASB/SCASW/SCASD-Compare String Data

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
AE	SCAS <span>m8</span>	X	X	X	X	X	X	Compare AL with byte at ES:[(E)DI], update [(E)DI]
AF	SCAS <span>m16</span>	X	X	X	X	X	X	Compare AX with word at ES:[(E)DI], update [(E)DI]
AF	SCAS <span>m32</span>				X	X	X	Compare EAX with dword at ES:[(E)DI], update [(E)DI]
AE	SCASB	X	X	X	X	X	X	Compare AL with byte at ES:[(E)DI], update [(E)DI]
AF	SCASW	X	X	X	X	X	X	Compare AX with word at ES:[(E)DI], update [(E)DI]
AF	SCASD				X	X	X	Compare EAX with dword at ES:[(E)DI], update [(E)DI]

## Description

The SCAS instruction subtracts the memory byte or word at the destination register from the AL, AX or EAX register. The result is discarded; only the flags are set. The operand must be addressable from the ES segment; no segment override is possible.

If the address-size attribute for this instruction is 16 bits, the DI register is used as the destination register, otherwise, the address-size attribute is 32 bits and the EDI register is used.

The address of the memory data being compared is determined solely by the contents of the destination register, not by the operand to the SCAS instruction. The operand validates ES segment addressability and determines the data type. Load the correct index value into the DI or EDI register before running the SCAS instruction.

After the comparison is made, the destination register is automatically updated. If the direction flag is 0 (the CLD instruction was run), the destination register is incremented; if the direction flag is 1 (the STD instruction was run), it is decremented. The increments or decrements are by 1 if bytes are compared, by 2 if words are compared, or by 4 if doublewords are compared.

The SCASB, SCASW, and SCASD instructions are synonyms for the byte, word and doubleword SCAS instructions that don't require operands. They are simpler to code, but provide no type or segment checking.

The SCAS instruction can be preceded by the REPE or REPNE prefix for a block search of CX or ECX bytes or words. Refer to the REP instruction for further details.

---

## Operation

```
IF AddressSize = 16 THEN use DI for dest-index; ELSE
E (* AddressSize = 32 *) use EDI for dest-index;
FI;
If byte type of instruction
THEN
  AL - [dest-index]; (* Compare byte in AL and dest *)
  IF DF = 0 THEN IncDec ` 1 ELSE IncDec ` -1; FI;
ELSE
  IF OperandSize = 16
  THEN
    AX - [dest-index]; (* compare word in AL and dest *)
    IF DF = 0 THEN IncDec ` 2 ELSE IncDec ` -2; FI;
  ELSE (* OperandSize = 32 *)
    EAX - [dest-index]; (* compare dword in EAX and dest *)
    IF DF = 0 THEN IncDec ` 4 ELSE IncDec ` -4; FI;
  FI;
FI;
dest-index = dest-index + IncDec
```

---

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
*			*	*	*	*	*

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

---

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the ES segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level

is 3.

# Related Information

- [Description](#)
- [Flags Affected](#)
- [Operation](#)
- [Protected Mode Exceptions](#)
- [Real Address Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

# SETcc-Byte Set on Condition

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F 97	SETA <a href="#">r/m8</a>				X	X	X	Set byte if above (CF=0 and ZF=0)
0F 93	SETAE <a href="#">r/m8</a>				X	X	X	Set byte if above or equal (CF=0)
0F 92	SETB <a href="#">r/m8</a>				X	X	X	Set byte if below (CF=1)
0F 96	SETBE <a href="#">r/m8</a>				X	X	X	Set byte if below or equal (CF=1 or ZF=1)
0F 92	SETC <a href="#">r/m8</a>				X	X	X	Set byte if carry (CF=1)
0F 94	SETE <a href="#">r/m8</a>				X	X	X	Set byte if equal (ZF=1)
0F 9F	SETG <a href="#">r/m8</a>				X	X	X	Set byte if greater (ZF=0 or SF=OF)
0F 9D	SETGE <a href="#">r/m8</a>				X	X	X	Set byte if greater or equal (SF=OF)
0F 9C	SETL <a href="#">r/m8</a>				X	X	X	Set byte if less (SF<>OF)
0F 9E	SETLE <a href="#">r/m8</a>				X	X	X	Set byte if less or equal (ZF=1 and SF<>OF)
0F 96	SETNA <a href="#">r/m8</a>				X	X	X	Set byte if not above (CF=1)
0F 92	SETNAE <a href="#">r/m8</a>				X	X	X	Set byte if not above or equal (CF=1)
0F 93	SETNB <a href="#">r/m8</a>				X	X	X	Set byte if not below (CF=0)
0F 97	SETNBE <a href="#">r/m8</a>				X	X	X	Set byte if not below or equal (CF=0 and ZF=0)
0F 93	SETNC <a href="#">r/m8</a>				X	X	X	Set byte if not carry (CF=0)
0F 95	SETNE <a href="#">r/m8</a>				X	X	X	Set byte if not equal (ZF=0)

0F 9E	SETNG <i>r/m8</i>	X X X Set byte if not greater (ZF=1 or SF<>OF)
0F 9C	SETNGE <i>r/m8</i>	X X X Set byte if not greater or equal (SF<>OF)
0F 9D	SETNL <i>r/m8</i>	X X X Set byte if not less (SF=OF)
0F 9F	SETNLE <i>r/m8</i>	X X X Set byte if not less or equal (ZF=1 and SF<>OF)
0F 91	SETNO <i>r/m8</i>	X X X Set byte if not overflow (OF=0)
0F 9B	SETNP <i>r/m8</i>	X X X Set byte if not parity (PF=0)
0F 99	SETNS <i>r/m8</i>	X X X Set byte if not sign (SF=0)
0F 95	SETNZ <i>r/m8</i>	X X X Set byte if not zero (ZF=0)
0F 90	SETO <i>r/m8</i>	X X X Set byte if overflow (OF=1)
0F 9A	SETP <i>r/m8</i>	X X X Set byte if parity (PF=1)
0F 9A	SETPE <i>r/m8</i>	X X X Set byte if parity even (PF=1)
0F 9B	SETPO <i>r/m8</i>	X X X Set byte if parity odd (PF=0)
0F 98	SETS <i>r/m8</i>	X X X Set byte if sign (SF=1)
0F 94	SETZ <i>r/m8</i>	X X X Set byte if zero (ZF=1)

## Description

The SETcc instruction stores a 1 byte at the destination specified by the effective address or register if the condition is met, or a 0 byte if the condition is not met.

## Operation

IF condition THEN *r/m8* ← 1 ELSE *r/m8* ← 0; FI;

## Protected Mode Exceptions

#GP(0) if the result is in a non-writable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

# Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault.

## Related Information

- Description
- Flags Affected
- Operation
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

# SGDT/SIDT-Store Global/Interrupt Descriptor Table Register

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F 01 /0	SGDT <i>m</i>			X	X	X	X	Store GDTR to <i>m</i> (6 bytes)
0F 01 /1	SIDT <i>m</i>			X	X	X	X	Store IDTR to <i>m</i> (6 bytes)

## Description

The SGDT and SIDT instructions copy the contents of the descriptor table register to the six bytes of memory indicated by the operand. The LIMIT field of the register is assigned to the first word at the effective address. If the operand-size attribute is 16 bits, the next three bytes are assigned the BASE field of the register, and the fourth byte is undefined. Otherwise, if the operand-size attribute is 32 bits, the next four bytes are assigned the 32-bit BASE field of the register.

The SGDT and SIDT instructions are used only in operating system software; they are not used in application programs.

## Operation

DEST ` 48-bit BASE/LIMIT register contents;

---

## Protected Mode Exceptions

Interrupt 6 if the destination operand is a register; #GP(0) if the destination is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 6 if the destination operand is a register; Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Compatibility Note

The 16-bit forms of the SGDT and SIDT instruction are compatible with the Intel 286 processor, if the value in the upper eight bits is not referenced. The Intel 286 processor stores 1's in these upper bits, whereas the 32-bit processors store 0's if the operand-size attribute is 16 bits. These bits were specified as undefined by the SGDT and SIDT instructions in the *80286 Programming Reference Manual* (Intel Order No. 210498).

---

## Related Information

[Compatibility Note](#)

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## SHL/SHR-Shift Instruction

See entry for SAL/SAR/SHL/SHR.

---

# SHLD-Double Precision Shift Left

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F A4	SHLD <i>r/m16,r16,imm8</i>	X	X	X				<i>r/m16</i> ` SHL or <i>r/m16</i> concatenated with <i>r16</i>
0F A4	SHLD <i>r/m32,r32,imm8</i>	X	X	X				<i>r/m32</i> ` SHL of <i>r/m32</i> concatenated with <i>r32</i>
0F A5	SHLD <i>r/m16,r16,CL</i>	X	X	X				<i>r/m16</i> ` SHL of <i>r/m16</i> concatenated with <i>r16</i>
0F A5	SHLD <i>r/m32,r32,CL</i>	X	X	X				<i>r/m32</i> ` SHL of <i>r/m32</i> concatenated with <i>r32</i>

---

## Description

The SHLD instruction shifts the first operand provided by the *r/m* field to the left as many bits as specified by the count operand. The second operand (*r16* or *r32*) provides the bits to shift in from the right (starting with bit 0). The result is stored back into the *r/m* operand. The register remains unaltered.

The count operand is provided by either an immediate byte or the contents of the CL register. These operands are taken MODULO 32 to provide a number between 0 and 31 by which to shift. Because the bits to shift are provided by the specified registers, the operation is useful for multi-precision shifts (64 bits or more). The SF, ZF and PF flags are set according to the value of the result. The CF flag is set to the value of the last bit shifted out. The OF and AF flags are left undefined.

---

## Operation

(\* count is an unsigned integer corresponding to the last operand of the instruction, either an immediate byte or the byte in register CL \*)

ShiftAmt ` count MOD 32;

inBits ` register; (\* Allow overlapped operands \*)

IF ShiftAmt = 0

THEN no operation

ELSE

IF ShiftAmt > OperandSize

THEN (\* Bad parameters \*)

*r/m* ` UNDEFINED;

CF, OF, SF, ZF, AF, PF ` UNDEFINED;

ELSE (\* Perform the shift \*)

CF ` BIT[Base, OperandSize - ShiftAmt];

(\* Last bit shifted out on exit \*)

FOR i ` OperandSize - 1 DOWNT0 ShiftAmt

DO

```

    BIT[Base, i] ` BIT[Base, i - ShiftAmt];
OF;
FOR i ` ShiftAmt - 1 DOWNT0 0
DO
    BIT[Base, i] ` BIT[inBits, i - ShiftAmt + OperandSize];
OD;
Set SF, ZF, PF (rm);
(* SF, ZF, PF are set according to the value of the result *)
AF ` UNDEFINED;
FI;
FI;

```

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
?			*	*	?	*	*

If count = 0, the flags are not affected.

The SF, ZF, and PF flags are set according to the result; the CF flag is set to the value of the last bit shifted out; after a shift of one bit position, the OF flag is set if a sign change occurred, otherwise it is cleared; after a shift of more than one bit position, the OF flag is undefined; the AF flag is undefined, except for a shift count of zero, which does not affect any flags.

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Related Information

[Description](#)

[Flags Affected](#)

# SHRD-Double Precision Shift Right

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F AC	SHRD <i>r/m16</i> , <i>r16</i> , <i>imm8</i>			X	X	X		<i>r/m16</i> ` SHR of <i>r/m16</i> concatenated with <i>r16</i>
0F AC	SHRD <i>r/m32</i> , <i>r32</i> , <i>imm8</i>			X	X	X		<i>r/m32</i> ` SHR of <i>r/m32</i> concatenated with <i>r32</i>
0F AD	SHRD <i>r/m16</i> , <i>r16</i> ,CL			X	X	X		<i>r/m16</i> ` SHR of <i>r/m16</i> concatenated with <i>r16</i>
0F AD	SHRD <i>r/m32</i> , <i>r32</i> ,CL			X	X	X		<i>r/m32</i> ` SHR of <i>r/m32</i> concatenated with <i>r32</i>

## Description

The SHRD instruction shifts the first operand provided by the *///m* field to the right as many bits as specified by the count operand. The second operand (*r16* or *r32*) provides the bits to shift in from the left (starting with bit 31). The result is stored back into the *///m* operand. The register remains unaltered.

The count operand is provided by either an immediate byte or the contents of the CL register. These operands are taken MODULO 32 to provide a number between 0 and 31 by which to shift. Because the bits to shift are provided by the specified register, the operation is useful for multi-precision shifts (64 bits or more). The SF, ZF and PF flags are set according to the value of the result. The CF flag is set to the value of the last bit shifted out. The OF and AF flags are left undefined.

## Operation

(\* count is an unsigned integer corresponding to the last operand of the instruction, either an immediate byte or the byte in register CL \*)  
ShiftAmt ` count MOD 32;  
inBits ` register; (\* Allow overlapped operands \*)  
IF ShiftAmt = 0  
THEN no operation  
ELSE

```

IF ShiftAmt > OperandSize
THEN (* Bad parameters *)
    //m ` UNDEFINED;
    CF, OF, SF, ZF, AF, PF ` UNDEFINED;
ELSE (* Perform the shift *)
    CF ` BIT[//m, ShiftAmt - 1]; (* last bit shifted out on exit *)
    FOR i ` 0 TO OperandSize - 1 - ShiftAmt
    DO
        BIT[//m, i] ` BIT[//m, i - ShiftAmt];
    OD;
    FOR i ` OperandSize - ShiftAmt TO OperandSize-1
    DO
        BIT[//m, i] ` BIT[inBits, i + ShiftAmt - OperandSize];
    OD;
    (* SF, ZF, PF are set according to the value of the result *)
    Set SF, ZF, PF (//m);
    AF ` UNDEFINED;
FI;
FI;

```

-----

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
?			*	*	?	*	*

If count = 0, the flags are not affected.

The SF, ZF, and PF flags are set according to the result; the CF flag is set to the value of the last bit shifted out; after a shift of one bit position, the OF flag is set if a sign change occurred, otherwise it is cleared; after a shift of more than one bit position, the OF flag is undefined; the AF flag is undefined, except for a shift count of zero, which does not affect any flags.

-----

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

-----

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Related Information

[Description](#)  
[Flags Affected](#)  
[Operation](#)  
[Protected Mode Exceptions](#)  
[Real Address Mode Exceptions](#)  
[Virtual 8086 Mode Exceptions](#)

---

# SIDT-Store Interrupt Descriptor Table Register

See entry for SGDT/SIDT.

---

# SLDT-Store Local Descriptor Table Register

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F 00 /0	SLDT <a href="#">r/m16</a>			<a href="#">X</a>	<a href="#">X</a>	<a href="#">X</a>	<a href="#">X</a>	Store LDTR to EA word

---

## Description

The SLDT instruction stores the Local Descriptor Table Register (LDTR) in the two-byte register or memory location indicated by the effective address operand. This register is a selector that points into the Global Descriptor Table.

The SLDT instruction is used only in operating system software. It is not used in application programs.

---

## Operation

*r/m16* ← LDTR;

---

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 6; the SLDT instruction is not recognized in Real Address Mode.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode (because the instruction is not recognized, it will not run or perform a memory reference).

---

## Notes

When the destination is a 32-bit register, the 16-bit source operand is copied into the lower 16 bits of the destination register, and the upper 16 bits of the register are undefined. With a 16-bit register operand, only the lower 16 bits of the destination are affected (the upper 16 bits remain unchanged). With a memory operand, the source is written to memory as a 16-bit quantity, regardless of operand size. As a result, 32-bit software should always treat the destination as 16-bits and mask bits 16-31, if necessary.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## SMSW-Store Machine Status Word

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F 01 /4	SMSW <i>r/m16</i>			X	X	X	X	Store machine status word to EA word

## Description

The SMSW instruction stores the machine status word (part of the CR0 register) in the two-byte register or memory location indicated by the effective address operand.

## Operation

*r/m16* ← MSW;

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode, #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Notes

This instruction is provided for compatibility with the Intel 286 processor; programs for the Pentium processor should use the MOV ..., CR0 instruction.

When the destination is a 32-bit register, the 16-bit source operand is copied into the lower 16 bits of the destination register, and the upper 16 bits of the register are undefined. With a 16-bit register operand, only the lower 16 bits of the destination are affected (the upper 16 bits remain unchanged). With a memory operand, the source is written to memory as a 16-bit quantity, regardless of operand size. As a result, 32-bit software should always treat the destination as 16-bits and mask bits 16-31, if necessary.

# Related Information

- Description
- Flags Affected
- Notes
- Operation
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

# STC-Set Carry Flag

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
F9	STC	X	X	X	X	X	X	Set carry flag

# Description

The STC instruction sets the CF flag.

# Operation

CF ← 1;

# Flags Affected

The CF flag is set.

## Related Information

- [Description](#)
- [Flags Affected](#)
- [Operation](#)
- [Protected Mode Exceptions](#)
- [Protected Mode Exceptions](#)
- [Virtual 8086 Mode Exceptions](#)

## STD-Set Direction Flag

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
FD	STD	X	X	X	X	X	X	Set direction flag so that [(E)SI] and/or [(E)DI] decrement

## Description

The STD instruction sets the direction flag, causing all subsequent string operations to decrement the index registers, (E)SI and/or (E)DI, on which they operate.

## Operation

DF ← 1;

---

# Flags Affected

OF DF IF SF ZF AF PF CF

1

The DF flag is set.

---

# Related Information

- Description
- Flags Affected
- Operation
- Protected Mode Exceptions
- Protected Mode Exceptions
- Virtual 8086 Mode Exceptions

---

# STI-Set Interrupt Flag

---

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
FB	STI	X	X	X	X	X	X	Set interrupt flag

---

# Description

The STI instruction sets the IF. The processor then responds to external interrupts after running the next instruction if the next instruction allows the IF flag to remain enabled. If external interrupts are disabled and the STI instruction is followed by the RET instruction (such as at the end of a subroutine), the RET instruction is allowed to run before external interrupts are recognized. Also, if external interrupts are disabled and the STI instruction is followed by a CLI instruction, which clears the IF flag, then external interrupts are not recognized because the CLI instruction clears the IF flag during its processing.

---

# Operation

```
IF PE=0 (* Running in real-address mode *)
THEN
  IF ` 1; (* Set Interrupt Flag *)
ELSE (* Running in protected mode or virtual-8086 mode *)
  IF VM=0 (* Running in protected mode *)
  THEN
    IF IOPL=3
    THEN IF ` 1; (* Set Interrupt Flag *)
    ELSE IF CPL IOPL
    THEN IF ` 1;
    ELSE #GP(0);
    FI;
  FI;
ELSE (* Running in Virtual-8086 mode *)
  #GP(0); (* Trap to virtual-8086 monitor *)
FI;
FI;
```

---

## Decision Table

The following decision table indicates which action in the lower portion of the table is taken given the conditions in the upper portion of the table.

PE =	0	1	1	1
VM =	-	0	0	1
CPL	-	IOPL > IOPL	= 3	
IOPL	-	-	-	= 3
IF ` 1	Y	Y		Y
#GP(0)			Y	

### Notes:

-	Don't care
Blank	Action not taken
Y	Action in Column 1 taken

---

## Flags Affected

OF DF IF SF ZF AF PF CF

The IF flag is set.

-----

## Protected Mode Exceptions

#GP(0) if the current privilege level is greater (has less privilege) than the I/O privilege level.

-----

## Virtual 8086 Mode Exceptions

#GP(0) as for protected mode.

-----

## Notes

In case of an NMI, trap, or fault following STI the interrupt will be taken before running the next sequential instruction in the code.

For information on this instruction when using virtual mode extensions, see the Intel documentation.

-----

## Related Information

- [Decision Table](#)
  - [Description](#)
  - [Flags Affected](#)
  - [Notes](#)
  - [Operation](#)
  - [Protected Mode Exceptions](#)
  - [Protected Mode Exceptions](#)
  - [Virtual 8086 Mode Exceptions](#)
- 

## STOS/STOSB/STOSW/STOSD-Store String Data

-----

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
----------	-------------	---	---	---	---	---	---	-------------

AA	STOS <b>m8</b>	X X X X X X	Store AL in byte at ES:[(E)DI], update [(E)DI]
AB	STOS <b>m16</b>	X X X X X X	Store AX in word at ES:[(E)DI], update [(E)DI]
AB	STOS <b>m32</b>	X X X	Store EAX in dword at ES:[(E)DI], update [(E)DI]
AA	STOSB	X X X X X X	Store AL in byte at ES:[(E)DI], update [(E)DI]
AB	STOSW	X X X X X X	Store AX in word at ES:[(E)DI], update [(E)DI]
AB	STOSD	X X X	Store EAX in dword at ES:[(E)DI], update [(E)DI]

## Description

The STOS instruction transfers the contents of the AL, AX, or EAX register to the memory byte or word given by the destination register relative to the ES segment. The destination register is the DI register for an address-size attribute of 16 bits or the EDI register for an address-size attribute of 32 bits.

The destination operand must be addressable from the ES register. A segment override is not possible.

The address of the destination is determined by the contents of the destination register, not by the explicit operand of the STOS instruction. This operand is used only to validate ES segment addressability and to determine the data type. Load the correct index value into the destination register before running the STOS instruction.

After the transfer is made, the (E)DI register is automatically updated. If the DF flag is 0 (the CLD instruction was run), the (E)DI register is incremented; if the DF flag is 1 (the STD instruction was executed), the (E)DI register is decremented. The (E)DI register is incremented or decremented by 1 if a byte is stored, by 2 if a word is stored, or by 4 if a doubleword is stored.

The STOSB, STOSW, and STOSD instructions are synonyms for the byte, word, and doubleword STOS instructions, that do not require an operand. They are simpler to use, but provide no type or segment checking.

The STOS instruction can be preceded by the REP prefix for a block fill of CX or ECX bytes, words, or doublewords. Refer to the REP instruction for further details.

## Operation

```

IF AddressSize = 16
THEN use ES:DI for DestReg
ELSE (* AddressSize = 32 *) use ES:EDI for DestReg;
FI;
IF byte type of instruction
THEN
  (ES:DestReg) ` AL;
  IF DF = 0
  THEN DestReg ` DestReg + 1;
  ELSE DestReg ` DestReg - 1;
  FI;
ELSE IF OperandSize = 16
THEN
  (ES:DestReg) ` AX;
  IF DF = 0
  THEN DestReg ` DestReg + 2;
  ELSE DestReg ` DestReg - 2;
  FI;
ELSE (* OperandSize = 32 *)
  (ES:DestReg) ` EAX;
  IF DF = 0

```

```
THEN DestReg ` DestReg + 4;
ELSE DestReg ` DestReg - 4;
FI;
FI;
FI;
```

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the ES segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

## STR-Store Task Register

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
----------	-------------	---	---	---	---	---	---	-------------

---

## Description

The contents of the task register are copied to the two-byte register or memory location indicated by the effective address operand.

The STR instruction is used only in operating system software. It is not used in application programs.

---

## Operation

*r/m* ` task register;

---

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 6; the STR instruction is not recognized in Real Address Mode.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode.

---

## Notes

When the destination is a 32-bit register, the 16-bit source operand is copied into the lower 16 bits of the destination register, and the upper 16 bits of the register are undefined. With a 16-bit register operand, only the lower 16 bits of the destination are affected (the upper 16 bits remain unchanged). With a memory operand, the source is written to memory as a 16-bit quantity, regardless of operand size. As a result, 32-bit software should always treat the destination as 16-bits and mask bits 16-31, if necessary.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

# SUB-Integer Subtraction

## Details Table

Encoding	Instruction	0 1 2 3 4 5	Description
2C ib	SUB AL,imm8	X X X X X X	AL ` AL - immediate byte
2D iw	SUB AX,imm16	X X X X X X	AX ` AX - immediate word
2D id	SUB EAX,imm32	X X X	EAX ` EAX - immediate dword
80 /5 ib	SUB r/m8,imm8	X X X X X X	r/m8 ` r/m8 - immediate byte
81 /5 iw	SUB r/m16,imm16	X X X X X X	r/m16 ` r/m16 - immediate word
81 /5 id	SUB r/m32,imm32	X X X	r/m32 ` r/m32 - immediate dword
83 /5 ib	SUB r/m16,imm8	X X X X X X	r/m16 ` r/m16 - sign-extended immediate byte
83 /5 ib	SUB r/m32,imm8	X X X	r/m32 ` r/m32 - sign-extended immediate byte
28 /r	SUB r/m8,r8	X X X X X X	r/m8 ` r/m8 - byte register
29 /r	SUB r/m16,r16	X X X X X X	r/m16 ` r/m16 - word register
29 /r	SUB r/m32,r32	X X X	r/m32 ` r/m32 - dword register
2A /r	SUB r8,r/m8	X X X X X X	r8 ` r8 - r/m byte
2B /r	SUB r16,r/m16	X X X X X X	r16 ` r16 - r/m word
2B /r	SUB r32,r/m32	X X X	r32 ` r32 - r/m dword

## Description

The SUB instruction subtracts the second operand (SRC) from the first operand (DEST). The first operand is assigned the result of the subtraction, and the flags are set accordingly.

When an immediate byte value is subtracted from a word operand, the immediate value is first sign-extended to the size of the destination operand.

-----

## Operation

IF SRC is a byte and DEST is a word or dword  
THEN DEST = DEST - SignExtend(SRC);  
ELSE DEST = DEST - SRC;  
FI;

-----

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
*			*	*	*	*	*

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

-----

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

-----

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

-----

## Related Information

[Description](#)

Flags Affected

Operation

Protected Mode Exceptions

Real Address Mode Exceptions

Virtual 8086 Mode Exceptions

# TEST-Logical Compare

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
A8 ib	TEST AL,imm8	X	X	X	X	X	X	AND immediate byte with AL
A9 iw	TEST AX,imm16	X	X	X	X	X	X	AND immediate word with AX
A9 id	TEST EAX,imm32				X	X	X	AND immediate dword with EAX
F6 /0 ib	TEST r/m8,imm8	X	X	X	X	X	X	AND immediate byte with r/m byte
F7 /0 iw	TEST r/m16,imm16	X	X	X	X	X	X	AND immediate word with r/m word
F7 /0 id	TEST r/m32,imm32				X	X	X	AND immediate dword with r/m dword
84 /r	TEST r/m8,r8	X	X	X	X	X	X	AND byte register with r/m byte
85 /r	TEST r/m16,r16	X	X	X	X	X	X	AND word register with r/m word
85 /r	TEST r/m32,r32				X	X	X	AND dword register with r/m dword

## Description

The TEST instruction computes the bit-wise logical AND of its two operands. Each bit of the result is 1 if both of the corresponding bits of the operands are 1; otherwise, each bit is 0. The result of the operation is discarded and only the flags are modified.

## Operation

DEST := LeftSRC AND RightSRC;  
CF `0;  
OF `0;

# Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
0			*	*	?	*	0

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result.

---

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## VERR, VERW-Verify a Segment for Reading or Writing

---

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F 00 /4	VERR <i>r/m16</i>		X	X	X	X	X	Set ZF=1 if segment can be read, selector in <i>r/m16</i>
0F 00 /5	VERW <i>r/m16</i>		X	X	X	X	X	Set ZF=1 if segment can be written, selector in <i>r/m16</i>

## Description

The two-byte register or memory operand of the VERR and VERW instructions contains the value of a selector. The VERR and VERW instructions determine whether the segment denoted by the selector is reachable from the current privilege level and whether the segment is readable (VERR) or writable (VERW). If the segment is accessible, the ZF flag is set; if the segment is not accessible, the ZF flag is cleared. To set the ZF flag, the following conditions must be met:

- The selector must denote a descriptor within the bounds of the table (GDT or LDT); the selector must be "defined."
- The selector must denote the descriptor of a code or data segment (not that of a task state segment, LDT, or a gate).
- For the VERR instruction, the segment must be readable. For the VERW instruction, the segment must be a writable data segment.
- If the code segment is readable and conforming, the descriptor privilege level (DPL) can be any value for the VERR instruction. Otherwise, the DPL must be greater than or equal to (have less or the same privilege as) both the current privilege level and the selector's RPL.

The validation performed is the same as if the segment were loaded into the DS, ES, FS, or GS register, and the indicated access (read or write) were performed. The ZF flag receives the result of the validation. The selector's value cannot result in a protection exception, enabling the software to anticipate possible segment access problems.

## Operation

```
IF segment with selector at (r/m) is accessible
  with current protection level
  AND ((segment is readable for VERR) OR
        (segment is writable for VERW))
THEN ZF ← 1;
ELSE ZF ← 0;
FI;
```

## Flags Affected

OF DF IF SF ZF AF PF CF

\*

The ZF flag is set if the segment is accessible, cleared if it is not.

## Protected Mode Exceptions

Faults generated by illegal addressing of the memory operand that contains the selector; the selector is not loaded into any segment register, and no faults attributable to the selector operand are generated.

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 6; the VERR and VERW instructions are not recognized in Real Address Mode.

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #AC for unaligned memory reference if the current privilege level is 3.

## Related Information

- Description
- Flags Affected
- Operation
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

## WAIT-Wait

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
----------	-------------	---	---	---	---	---	---	-------------

---

## Description

WAIT causes the processor to check for pending unmasked numeric exceptions before proceeding.

---

## Protected Mode Exceptions

#NM if both MP and TS in CR0 are set.

---

## Real Address Mode Exceptions

Interrupt 7 if both MP and TS in CR0 are set.

---

## Virtual 8086 Mode Exceptions

#NM if both MP and TS in CR0 are set.

---

## Notes

Coding WAIT after an ESC instruction ensures that any unmasked floating-point exceptions the instruction may cause are handled before the processor has a chance to modify the instruction's results.

FWAIT is an alternate mnemonic for WAIT.

Refer to the Intel documentation for more information about when to use WAIT (FWAIT).

---

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

# WBINVD-Write-Back and Invalidate Cache

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F 09	WBINVD					X	X	Write-back and invalidate entire cache

---

## Description

The internal cache is flushed, and a special-function bus cycle is issued, which indicates that external cache should write-back its contents to main memory. Another special-function bus cycle follows, directing the external cache to flush itself.

---

## Operation

FLUSH INTERNAL CACHE  
SIGNAL EXTERNAL CACHE TO WRITE-BACK  
SIGNAL EXTERNAL CACHE TO FLUSH

---

## Protected Mode Exceptions

The WBINVD instruction is a privileged instruction; #GP(0) if the current privilege level is not 0.

---

## Virtual 8086 Mode Exceptions

#GP(0); the WBINVD instruction is a privileged instruction.

---

## Notes

INVD should be used with care. It does not write back modified cache lines; therefore, it can cause the data cache to become inconsistent

with other memories in the system. Unless there is a specific requirement or benefit to invalidate a cache without writing back the modified lines (i.e., testing or fault recovery where cache coherency with main memory is not a concern), software should use the WBINVD instruction.

This instruction is implementation-dependent; its function may be implemented differently on future Intel processors.

It is the responsibility of hardware to respond to the external cache write-back and flush indications.

This instruction is not supported on Intel386 processors. See the Intel documentation for information on detecting the processor type at runtime. See the Intel documentation for information on disabling the cache.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Protected Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## WRMSR-Write to Model Specific Register

---

### Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F 30	WRMSR							<b>X</b> Write the value in EDX:EAX to Model Specific Register indicated by ECX

---

### Description

The value in ECX specifies one of the 64-bit Model Specific Registers of the Pentium processor. The contents of EDX:EAX is copied into that Model-Specific Register. The high-order 32 bits are copied from EDX and the low-order 32 bits are copied from EAX.

The following values are used to select model specific registers on the Pentium processor:

VALUE	REGISTER NAME	DESCRIPTION
00H	Machine Check Address	Stores address of cycle causing the exception

01H Machine Check Type Stores cycle type of cycle causing the exception

For other values used to perform cache, TLB, and BTB testing and performance monitoring, see the Intel documentation.

---

## Operation

MSR[ECX] ← EDX:EAX;

---

## Protected Mode Exceptions

#GP(0) if either the current privilege level is not 0 or the value in ECX does not specify a Model-Specific Register that is implemented in the Pentium processor.

---

## Real Address Mode Exceptions

#GP(0) if the value in ECX does not specify a Model-Specific Register that is implemented in the Pentium processor. No error code is pushed.

---

## Virtual 8086 Mode Exceptions

#GP(0) if instruction execution is attempted.

---

## Notes

Always set undefined or reserved bits to the value previously read.

WRMSR is used to write the content of Model-Specific Registers that control functions for testability, execution tracing, performance monitoring, and machine check errors. Refer to the *Pentium(TM) Processor Data Book* for more information.

The values 3H, 0FH, and values above 13H are reserved. Do not execute WRMSR with reserved values in ECX.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

# XADD-Exchange and Add

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
0F C0 /r	XADD r/m8,r8					X	X	Exchange byte register and r/m byte; load sum into r/m byte
0F C1 /r	XADD r/m16,r16					X	X	Exchange word register and r/m word; load sum into r/m word
0F C1 /r	XADD r/m32,r32					X	X	Exchange dword register and r/m dword; load sum into r/m dword

## Description

The XADD instruction loads DEST into SRC, and then loads the sum of DEST and the original value of SRC into DEST.

## Operation

TEMP ` SRC + DEST  
SRC ` DEST  
DEST ` TEMP

## Flags Affected

OF DF IF SF ZF AF PF CF  
\* \* \* \* \*

The CF, PF, AF, SF, ZF, and OF flags are affected as if an ADD instruction had been executed.

---

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #NM if either EM or TS in CR0 is set, #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in real-address mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Notes

This instruction can be used with a LOCK prefix. The Intel386 DX microprocessor does not implement this instruction. If this instruction is used, you should provide an equivalent code sequence that runs on an Intel386 DX processor as well. See the Intel documentation for details on how to detect a particular processor at runtime.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Notes](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## XCHG-Exchange Register/Memory with Register

---

# Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
90+rw	XCHG AX, <span style="color: blue;">r16</span>	X	X	X	X	X	X	Exchange word register with AX
90+rd	XCHG EAX, <span style="color: blue;">r32</span>			X	X	X		Exchange dword register with EAX
90+rw	XCHG <span style="color: blue;">r16</span> , AX	X	X	X	X	X	X	Exchange word register with AX
90+rd	XCHG <span style="color: blue;">r32</span> , EAX			X	X	X		Exchange dword register with EAX
86 /r	XCHG <span style="color: blue;">r/m8</span> , r8	X	X	X	X	X	X	Exchange byte register with EA byte
86 /r	XCHG <span style="color: blue;">r8</span> , <span style="color: blue;">r/m8</span>	X	X	X	X	X	X	Exchange byte register with EA byte
87 /r	XCHG <span style="color: blue;">r/m16</span> , <span style="color: blue;">r16</span>	X	X	X	X	X	X	Exchange word register with EA word
87 /r	XCHG <span style="color: blue;">r/m32</span> , <span style="color: blue;">r32</span>			X	X	X		Exchange dword register with EA dword
87 /r	XCHG <span style="color: blue;">r16</span> , <span style="color: blue;">r/m16</span>	X	X	X	X	X	X	Exchange word register with EA word
87 /r	XCHG <span style="color: blue;">r32</span> , <span style="color: blue;">r/m32</span>			X	X	X		Exchange dword register with EA dword

## Description

The XCHG instruction exchanges two operands. The operands can be in either order. If a memory operand is involved, the LOCK# signal is asserted for the duration of the exchange, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL.

## Operation

temp ← DEST  
DEST ← SRC  
SRC ← temp

## Protected Mode Exceptions

#GP(0) if either operand is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

**Note**

XCHG can be used for BSWAP for 16-bit data.

## Related Information

- Description
- Flags Affected
- Operation
- Protected Mode Exceptions
- Real Address Mode Exceptions
- Virtual 8086 Mode Exceptions

## XLAT/XLATB-Table Look-up Translation

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
D7	XLAT <span>m8</span>	X	X	X	X	X	X	Set AL to memory byte DS:[(E)BX + unsigned AL]
D7	XLATB	X	X	X	X	X	X	Set AL to memory byte DS:[(E)BX + unsigned AL]

## Description

The XLAT instruction changes the AL register from the table index to the table entry. The AL register should be the unsigned index into a table addressed by the DS:BX register pair (for an address-size attribute of 16 bits) or the DS:EBX register pair (for an address-size attribute of 32

bits).

The operand to the XLAT instruction allows for the possibility of a segment override. The XLAT instruction uses the contents of the BX register even if they differ from the offset of the operand. The offset of the operand should have been moved into the BX or EBX register with a previous instruction.

The no-operand form, the XLATB instruction, can be used if the BX or EBX table will always reside in the DS segment.

---

## Operation

```
IF AddressSize = 16
THEN
  AL ← (BX + ZeroExtend(AL))
ELSE (* AddressSize = 32 *)
  AL ← (EBX + ZeroExtend(AL));
FI;
```

---

## Protected Mode Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

---

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

---

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

---

## XOR-Logical Exclusive OR

---

## Details Table

Encoding	Instruction	0	1	2	3	4	5	Description
34 ib	XOR AL,imm8	X	X	X	X	X	X	Exclusive-OR immediate byte to AL
35 iw	XOR AX,imm16	X	X	X	X	X	X	Exclusive-OR immediate word to AX
35 id	XOR EAX,imm32				X	X	X	Exclusive-OR immediate dword to EAX
80 /6 ib	XOR r/m8,imm8	X	X	X	X	X	X	Exclusive-OR immediate byte to r/m byte
81 /6 iw	XOR r/m16,imm16	X	X	X	X	X	X	Exclusive-OR immediate word to r/m word
81 /6 id	XOR r/m32,imm32				X	X	X	Exclusive-OR immediate dword to r/m dword
83 /6 ib	XOR r/m16,imm8				X	X	X	XOR sign-extended immediate byte to r/m word
83 /6 id	XOR r/m32,imm8				X	X	X	XOR sign-extended immediate byte with r/m dword
30 /r	XOR r/m8,r8	X	X	X	X	X	X	Exclusive-OR byte register to r/m byte
31 /r	XOR r/m16,r16	X	X	X	X	X	X	Exclusive-OR word register to r/m word
31 /r	XOR r/m32,r32				X	X	X	Exclusive-OR dword register to r/m dword
32 /r	XOR r8,r/m8	X	X	X	X	X	X	Exclusive-OR r/m byte to byte register
33 /r	XOR r16,r/m16	X	X	X	X	X	X	Exclusive-OR r/m word to word register
33 /r	XOR r32,r/m32				X	X	X	Exclusive-OR r/m dword to dword register

---

## Description

The XOR instruction computes the exclusive OR of the two operands. Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same. The answer replaces the first operand.

---

## Operation

DEST ← LeftSRC XOR RightSRC  
CF ← 0

## Flags Affected

OF	DF	IF	SF	ZF	AF	PF	CF
0			*	*	?	*	0

The CF and OF flags are cleared; the SF, ZF, and PF flags are set according to the result; the AF flag is undefined.

## Protected Mode Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Real Address Mode Exceptions

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.

## Virtual 8086 Mode Exceptions

Same exceptions as in Real Address Mode; #PF(fault-code) for a page fault; #AC for unaligned memory reference if the current privilege level is 3.

## Related Information

[Description](#)

[Flags Affected](#)

[Operation](#)

[Protected Mode Exceptions](#)

[Real Address Mode Exceptions](#)

[Virtual 8086 Mode Exceptions](#)

## Assembler Messages

This section describes all the messages produced by ALP at run time. Messages issued by the assembler have the following format:

**[Coordinates] [Severity Type] [Message Number] [Message Content]**

The following sections describe the various fields common to all assembler messages, and a complete description of each individual message is included.

---

## Message Coordinates

Message coordinates (if present) appear as the first field within a message, and have one of two forms:

**[Path]Filename(LLLL:CCCC):**

**[Path]Filename(LLLL,Macroname(LLLL,CCCC)):**

Coordinates are displayed as part of a message if ALP is parsing an input stream and the event which caused the message to be displayed is directly related to a specific location in the input. The coordinates show the user exactly where to look if action is required. The fact the ALP is parsing an input stream does not mean that coordinates will appear in a message; some messages may occur during parsing that are not a reference to the input stream.

- **Filename**

This is the name of the file containing the input token which caused the message to be generated. If this is the root source file whose name was passed on the assembler command line, the file name will be displayed exactly as specified by the user. If this is an **INCLUDE** file, it will be displayed exactly as specified in the **INCLUDE** directive, and the path name where the file was searched for and found (if any) will be prepended to the beginning. ALP does not query the operating system in an attempt to derive the full path name of a partially qualified file.

- **Macroname**

If the assembler is currently parsing tokens within a macro expansion, the name of the macro currently being expanded will appear in the coordinates.

- **Line Number (LLLL)**

The first number in parentheses is the line number within the source file where the referenced token is located; this refers to the *outer-most point of invocation* if a macro name is also given in the message coordinates. A line number value appearing within parentheses following a macro name refers to the *innermost point of expansion* (since macro expansions may be nested) and references the original definition of the macro.

- **Column Number (CCCC)**

The second number in parentheses is the column number of the first character of the referenced token within the source file or macro definition.

---

## Message Severity Types

Every message displayed by ALP is assigned a specific type, and the type of message dictates the severity level. The following is a list of message types produced by ALP showing the type name (as it appears in the actual message), followed by a description of what caused the message, how severe it is, and the action taken by ALP after the message is generated.

<b>Info</b>	Informational message only; processing continues normally.
<b>Warning</b>	Questionable syntax or semantics; input file may be incorrect, but processing continues.
<b>Error</b>	Syntax or semantic error in input; continue processing, object output file is discarded.
<b>Fatal</b>	Unrecoverable user or environment error; terminate assembly prematurely after releasing resources and closing files.
<b>Internal</b>	Internal program logic error, abort immediately.
<b>Usage</b>	Incorrect command line syntax, abort.

When the assembler begins processing, the display of all warning messages is enabled; informational messages do not display unless they are requested. The display of both warning and informational messages may be controlled with the command line option [M - Control Individual Messages or Groups](#) See [Message Control Options](#) for more information on the behavior of assembler messages.

---

## Message Numbers and Message Content

Messages numbers displayed by ALP have the following format:

### **ALPnnnn:**

Message numbers always have a three-letter prefix that designates the component issuing the message (**ALP**), followed by a four digit decimal number given by **nnnn**. All messages issued by the assembler are uniquely numbered; however, not all messages displayed by the assembler will be accompanied by a formatted message number (for instance, the assembler banner).

Messages issued during assembler initialization, command line processing, or exception handling are numbered from 0 to 999. Other messages occur during input stream processing and are grouped according to their severity type: 1000 through 1999 for fatal errors, 3000 through 3999 for regular errors, 4000 through 4999 for warnings, and 5000 through 5999 for informational messages.

It should be noted that messages are numbered for reference only; it is not guaranteed that messages will be numbered identically for each subsequent assembler release, or that individual messages will be retained or remain unmodified in future releases.

---

## Message Numbers 0-999: Internal, Usage, and Special Case Messages

Messages in this section normally occur during assembler initialization, when errors are encountered during command line processing, or when exceptional conditions occur that prevent the assembler from completing initialization or execution.

---

### **ALP0004: received, is shutting down**

ALP has handled a request from the operating system to abort execution. The type of abort request is noted in the text of the message. All open files will be closed and any incomplete output files will be deleted.

**Recovery:** If termination was requested by the user, no further action is necessary. Otherwise, the operating system may have sent an abort signal because of insufficient system resources.

---

### **ALP0005: Assertion failure,**

This message is displayed when an internal self-check condition has been violated, and indicates an error in the internal programming logic of the assembler. This message should never occur.

**Recovery:** Note the conditions under which the error occurred, and if possible isolate a minimal test case that will reproduce the problem. Contact IBM.

---

### **ALP0942: -Lo:xxxxxxxxxxxx must be one each of "XYZLMICFOGD**

This option specifies the sort order for the individual vertical listing file columns. Not all single character tags that uniquely identify each individual column were specified.

**Recovery:** All column tags must be specified in the argument field of this option, even when the display of one or more columns has been disabled.

---

## ALP0981: Invalid or missing include path

The list of INCLUDE file directories was incorrectly specified.

---

## ALP0991: Invalid option ""

The command line parser encountered a character sequence on the command line that was interpreted as an option, but the option identifier itself was not recognized.

---

## ALP0992: Option "" not valid in global scope

An attempt was made to use an option in a situation that would cause ambiguities. As coded by the user, the option is only legal when applied to an individual filename.

**Recovery:** If the option syntax is correct, insure that it follows the filename to which it applies.

---

## ALP0993: Option "" not valid in local scope

An attempt was made to apply a global assembler option to an individual file.

**Recovery:** Global options must appear before any filenames; in most cases the usage of global options and filenames is a mutually exclusive operation.

---

## ALP0994: Invalid argument in option ""

In the argument field of a parameterized command line option, an argument of a specific type was expected, but an invalid token was encountered instead.

---

## ALP0995: Expecting ":" or "=" in option ""

A parameterized option was encountered, but no colon (:) or equal sign (=) followed the option identifier.

**Recovery:** Parameterized options must be immediately followed by a colon or equal sign with no intervening white space characters, followed by the option argument(s).

---

## ALP0996: Invalid message number

An explicit message number specified with the **-M** option did not identify a message for which switching is enabled.

**Recovery:** Only warning and informational messages may switched on or off.

---

## ALP0997: Invalid keyword "" in option ""

The referenced identifier was not a valid keyword; a keyword was expected in the context of the referenced option.

---

## Message Numbers 1000-1999: Fatal Error Messages

Fatal errors typically occur when the assembler requests a resource from the operating system, but the request fails. This may or may not be due to user error, but the assembler was unable to correct the problem and execution is terminated after an orderly shutdown is performed.

---

## ALP1101: Memory allocation error

The assembler attempted to dynamically allocate a block of storage, but the request was denied.

**Recovery:**

- Close any large or memory intensive processes
  - Verify that sufficient paging space exists
  - The host computer may have insufficient hardware resources
- 

## ALP1102: Too many error messages

This message is displayed when the assembler has reached the error limit threshold.

**Related Information:**

- [Me - Set Number of Errors Before Assembler Aborts](#)
- 

## ALP1103: Error opening message output file "";

An error occurred while attempting to open the referenced file.

**Recovery:** Verify that the file exists and that read permission is allowed. Verify that no other processes are accessing the file, and that the file system is functioning correctly.

---

## ALP1104: Input and message output filenames are identical

The assembler has detected an attempt to create an output file with a name identical to that of an input file; the operation was not allowed. The assembler only detects this condition when the names are an identical match, using a case-sensitive comparison algorithm.

**Recovery:** Ensure that the correct command line options have been used. Internal variables may have been incorrectly initialized using options that affect automatic file name generation, thus causing a filename collision.

Related Information:

- [Internal Variables](#)
  - [File Control Options](#)
- 

## ALP1401: Error opening listing output file "";

An error occurred while attempting to open the referenced file.

**Recovery:** Verify that the file exists and that read permission is allowed. Verify that no other processes are accessing the file, and that the file system is functioning correctly.

---

## ALP1402: Error writing listing output file "";

An error occurred while attempting to write to the referenced file.

**Recovery:** Ensure that there is sufficient space on the target file system, and no other processes are accessing or modifying the file. Verify that the file system is functioning correctly.

---

## ALP1403: Input and listing output filenames are identical

The assembler has detected an attempt to create an output file with a name identical to that of an input file; the operation was not allowed. The assembler only detects this condition when the names are an identical match, using a case-sensitive comparison algorithm.

**Recovery:** Ensure that the correct command line options have been used. Internal variables may have been incorrectly initialized using options that affect automatic file name generation, thus causing a filename collision.

Related Information:

- [Internal Variables](#)
  - [File Control Options](#)
- 

## ALP1601: Error creating object output file "";

An error occurred while attempting to create the referenced file.

**Recovery:** Verify that the target drive and directory exist and that create and write permission have been granted. Verify that no other processes are accessing the file, and that the file system is functioning correctly.

---

## ALP1602: Error writing object output file "";

An error occurred while attempting to write to the referenced file.

**Recovery:** Ensure that there is sufficient space on the target file system, and no other processes are accessing or modifying the file. Verify that the file system is functioning correctly.

---

## ALP1603: Input and object output filenames are identical

The assembler has detected an attempt to create an output file with a name identical to that of an input file; the operation was not allowed. The assembler only detects this condition when the names are an identical match, using a case-sensitive comparison algorithm.

**Recovery:** Ensure that the correct command line options have been used. Internal variables may have been incorrectly initialized using options that affect automatic file name generation, thus causing a filename collision.

Related Information:

- [Internal Variables](#)
- [File Control Options](#)

---

## ALP1801: Error opening source file "";

An error occurred while attempting to open the referenced file.

**Recovery:** Verify that the file exists and that read permission is allowed. Verify that no other processes are accessing the file, and that the file system is functioning correctly.

---

## ALP1802: Error reading source file "";

An error occurred while attempting to read from the referenced file.

**Recovery:** Ensure that the file exists, that the filelength is non-zero, and that read permission is allowed. Verify that no other processes are accessing the file, and that the file system is functioning correctly.

---

## ALP1803: Circular text substitution

The preprocessor has detected an attempt to perform a recursive text substitution operation, such as an INCLUDE file "including" itself, or a macro expanding itself.

---

## ALP1804: Internal buffer overflow

The preprocessor has overflowed an internal memory buffer while trying to process the referenced token.

**Recovery:** Reduce the number of whitespace characters preceding the token or the number of characters in the token itself.

---

## ALP1805: Input and dependency output filenames are identical

The assembler has detected an attempt to create an output file with a name identical to that of an input file; the operation was not allowed. The assembler only detects this condition when the names are an identical match, using a case-sensitive comparison algorithm.

**Recovery:** Ensure that the correct command line options have been used. Internal variables may have been incorrectly initialized using options that affect automatic file name generation, thus causing a filename collision.

Related Information:

- [Internal Variables](#)
- [File Control Options](#)

---

## ALP1901: Unexpected character in identifier

The command line parser was expecting an identifier but immediately encountered a non-identifier character. Command line parsing was prematurely terminated.

---

## ALP1902: Unexpected character in keyword

The command line parser was expecting an keyword but immediately encountered a non-identifier character. Command line parsing was prematurely terminated.

---

## ALP1903: Error opening response file "";

An error occurred while attempting to open the referenced file.

**Recovery:** Verify that the file exists and that read permission is allowed. Verify that no other processes are accessing the file, and that the file system is functioning correctly.

---

## ALP1904: Invalid filename in "@" directive

The command line parser was processing an @Filename (command line response file) directive, but the token following the "@" character did not constitute a valid filename.

**Recovery:** Ensure that only valid filename characters are used.

---

## ALP1905: Unexpected character or terminator

The command line parser encountered either an illegal control character or the end of the command line input stream before the current command parameter was completely parsed.

---

## ALP1906: Numeric constant is invalid;

An error occurred while converting the referenced numeric constant to an internal representation.

---

## Message Numbers 3000-3999: Error Messages

Error messages are typically issued during processing of the input stream and indicate a syntax or semantic error in the user program. The assembler will continue processing after an error has occurred, but since the input stream was incorrect an output object file will not be created.

---

## ALP3201: Can't

This message appears during expression processing when a binary operation was performed on two primary expressions of incompatible type. The message replacement parameters indicate the attempted operation.

---

## ALP3202: Overflow or division by zero

Either the expression evaluated to a quantity that is not representable in the current expression word size, or an expression contained a binary division (/) or modulus (MOD) operation where the denominator expression was evaluated to be zero.

---

## ALP3203: Can't take offset of expression

The expression to which the OFFSET operator was applied did not contain a constant or relocatable address.

**Recovery:** The offset operator may not be applied to register values. If the offset expression is applied to a quoted string, ensure that the string length does not exceed the length of what is representable as a constant value, given the word size of the enclosing segment (2 for USE16 segments, 4 for USE32 segments).

---

## ALP3204: Expecting relocatable expression

An expression was used in a context that required a segment or group relative address, but one was not supplied.

---

## ALP3205: Expecting primary expression

The assembler was expecting a terminal operand (an identifier, register, or constant) or an expression enclosed in parentheses () or square brackets []; instead, an unexpected token was encountered at the referenced location.

---

## ALP3206: Expecting "]"

An opening bracket "[" was encountered and the subsequent expression was fully parsed, but a closing bracket "]" was not encountered.

---

## ALP3207: Expecting ")"

An opening parenthesis "(" was encountered and the subsequent expression was fully parsed, but a closing parenthesis ")" was not

encountered.

---

## ALP3208: Forward reference needs segment override or FAR PTR

An expression contained a forward reference to a location that was later determined to be of FAR distance. When forward references are used, the assembler makes default assumptions about the eventual definition of undefined labels used in the expression; such definitions are never assumed to be in a different segment. Use of such an expression can cause differences between the code generated on the first and second passes of the assembler.

**Recovery:** Qualify the expression with a distance override (FAR, FAR16, or FAR32) or segment override (:) operator.

---

## ALP3210: Illegal operation on relocatable value

A unary operator was applied to an expression containing a relocatable address or indirect memory expression, but the operator may only be used with constant values.

---

## ALP3211: Illegal type expression

A "<type> PTR" type conversion expression was encountered, but the expression given by <type> was not a valid qualified type.

---

## ALP3212: Operands must be relative to same segment or group

A binary operation was performed on two relocatable expressions, but the expressions were not declared relative to the same segment.

---

## ALP3213: Illegal digit(s) for current .RADIX

A literal integer constant was not qualified with a prefix or suffix (radix override), but was specified using digits that are not valid for the current radix. For instance, use of the literal **1234** is illegal when the current radix value is 2.

---

## ALP3214: Value cannot be negative

A negative-value expression was encountered where only positive numbers are allowed, such as the count-value for the DUP operator, the field-width entry in a RECORD definition, or the shift-value in the SHL or SHR operators.

---

## ALP3215: Expression cannot be forward-referenced

The referenced expression is used in a context where an undefined or forward-referenced value prevents the assembler from completing the operation or generating a reliable construct. This is the case during the declaration of user-defined types or where the value of the location counter will depend on the results of the expression.

---

## ALP3216: Expression does not have an operand size

The SIZE operator was used on an expression for which no operand size exists (a simple number value, for example). The SIZE operator may only be used on expressions that have an operand size attribute, such as a variable name or type name expression.

---

## ALP3217: Record tag expected

A left-hand identifier operand was followed by a right-hand expression list operand enclosed in angle brackets or braces. The construct was parsed as a record constant, but could not be evaluated because the left-hand operand was not a record tag name previously defined with the RECORD directive.

---

## ALP3218: Numeric constant is invalid;

An error occurred while converting the referenced numeric constant to an internal representation.

---

## ALP3219: Expression must be a constant value

The assembler issues this message whenever a constant expression is required, but the expression supplied contained relocation information or machine register references.

---

## ALP3220: Undefined symbol ""

This message appears when an identifier was referenced in an expression or type declaration, and the identifier has no external declaration or definition or is forward-referenced.

---

## ALP3221: Expecting ", " or ")"

This message appears when processing a DUP expression list and an unexpected token was encountered. The parser was expecting the list to be continued with a comma or terminated with a closing parentheses.

**Recovery:** Check for possible unbalanced parentheses ().

---

## ALP3222: Expecting "("

The DUP operator was encountered but was not followed by an opening parentheses to begin the duplicated expression list.

**Recovery:** The duplicated expression list following the DUP operator must be enclosed in parentheses (), even if the expression list only contains a single item.

---

## ALP3223: Expecting variable name

The LENGTH operator may only be applied to data labels (variable names). LENGTH returns the number of items allocated to the data label when the label was defined in a data allocation statement.

-----

## ALP3224: Expecting ",", or ""

This message appears within a bracketed expression list where the assembler was expecting a comma to introduce the next initializer expression, or a closing brace or angle bracket to terminate the initializer expression list.

-----

## ALP3225: Register-indirect expression illegal in this context

A constant or relocatable address expression was expected but the expression also contained at least one machine register. Such an address may be calculated only at run time, and is illegal in this context.

-----

## ALP3226: Expression must have structure or union type

The expression to the left of a structure member selection operator (.) did not have a structure or union type. The assembler cannot evaluate the member expression to the right of the selection operator unless the left hand operand has an associated structure or union type.

**Recovery:** If the left hand operand must be something other than an explicit structure or union variable (such as a register-indirect expression like [EBX]), then use the PTR operator to convert the left hand operand to the appropriate type.

-----

## ALP3227: Right operand is not a member of structure or union

The expression to the right of a structure member selection operator (.) was not a member of the left-hand structure or union expression.

**Recovery:** The right-hand operand must be an identifier named in the type definition for the structure or union.

-----

## ALP3228: Invalid identifier type for this operation

The symbol type of the referenced identifier was not valid in this context.

**Recovery:** This could happen if a text macro name was used in a context where macro expansions are not performed.

-----

## ALP3229: Expression type not valid in this context

The referenced expression evaluated to a type that is not valid for the context in which it was used. An example of this would be the use of a segment name or group name where a data label was expected.

**Recovery:** Review the expression-types allowed by the statement or clause where the referenced expression was used, and modify the

construct accordingly.

---

## ALP3301: Label must be followed by a directive

A user identifier appeared in the label field, but the end of line was encountered before an assembler directive was specified to give the label a definition.

**Recovery:** Code labels must be followed by a single colon (:) or double colon (::) on the same line as the label itself. Named assembler directives must appear on the same line as the associated label, or the line continuation (\) character must follow the label to create a single logical line.

Check for a possible misspelled identifier or keyword. Verify that the identifier was specified using the correct uppercase and lowercase letters if case sensitive assembly is in effect.

---

## ALP3302: Expecting label, directive, or mnemonic

The token referenced in the error message was unexpected. This error occurs when the first token on the line is not a valid label, directive, or mnemonic, or when a valid label has been encountered but was not followed by a valid directive or mnemonic.

**Recovery:** If this message references an identifier, check that it was spelled correctly, or that the identifier was specified using the correct uppercase and lowercase letters if case sensitive assembly is in effect.

---

## ALP3303: Can't be preceded by data label

The referenced token is either an instruction mnemonic or a user identifier that appears after a label.

If the referenced token is an instruction mnemonic, then the preceding label must be followed by a single colon (:) or double colon (::). Data labels may not be used to refer to instructions. Otherwise, it is invalid to have two labels appearing in succession.

**Recovery:** Check for misspellings in either identifier, and that the identifiers were specified using the correct uppercase and lowercase letters if case sensitive assembly is in effect.

---

## ALP3501: Address size mismatch

This message indicates one of the following:

- An indirect memory expression contained a mixture of 16-bit or 32-bit base or index registers. This prevents the assembler from determining the address size of the expression, and is illegal.
  - The instruction performs an unalterable implicit operation which conflicts with the operands supplied.
- 

## ALP3502: Invalid register expression

A processor register was specified using indexed notation (i.e., "REG(X)"), but the expression in parentheses was out of range and did not refer to a valid register.

---

## ALP3503: Can't use this register with scale factor

In an indirect memory expression, the scaling operator (\*) was applied to a register for which the processor does not support a scaling operation.

A scaling factor may only be used on 80386 or later processors, and only in 32-bit address expressions. Within this context, only the EAX, EBX, ECX, EDX, EDI, ESI, and EBP registers are valid. A scaling factor may not be applied to the ESP register.

---

## ALP3504: Illegal target of self relative pointer

The SHORT operator was used, but the expression to which it was applied was not a simple self relative displacement. This can occur if the expression is a constant, data label, or indirect memory operand.

---

## ALP3505: Invalid mnemonic/operand combination

This message appears when a valid mnemonic has been recognized and all operand expressions have been correctly parsed, but the assembler was unable to combine the results into a form that it could associate with a valid instruction encoding.

**Recovery:** This usually indicates that one or more operand expressions were not correctly specified. Verify such factors as:

- Correctly specified operand sizes
- Register combinations allowable for this instruction
- Labels or identifiers are of the correct type
- Correct number of operands

---

## ALP3506: Register combination invalid with -bit expression

For 80386 processors or greater, both 16-bit and 32-bit effective addresses are supported, but the two modes differ in the register combinations that are allowed for indirect addressing. The expression used a combination that was invalid for the referenced address size.

For expressions that refer to 16-bit (USE16) memory locations, only a single base register (BX or BP) and/or a single index register (DI or SI) may be used.

For expressions that refer to 32-bit (USE32) memory locations, only a single base register (EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI) and/or a single index register (EAX, EBX, ECX, EDX, EBP, EDI, ESI) may be used; no single register may appear more than once in a given expression.

---

## ALP3507: Scaling factor must be 1, 2, 4, or 8

The processor only supports the scaling factors referenced in the message.

---

## ALP3508: Invalid use of register

One of the following illegal conditions occurred:

- An attempt was made to use an unsupported register in an indirect memory expression (for example, [AH]).

- An attempt was made to combine a register with other terms to form an indirect memory expression, and the register was not enclosed in square brackets [ ].
- A register argument was expected and correctly parsed in an assembler directive, but the referenced register cannot be legally used in this particular context.

-----

## ALP3509: Multiple base registers

An indirect memory expression contained a combination of registers that was invalid for the selected addressing mode. More than one register was evaluated as a **base register** by the assembler; the processor only supports a single base register within **register indirect** addressing mode.

For expressions that refer to 16-bit (USE16) memory locations, only **BX** and **BP** are valid base registers.

For expressions that refer to 32-bit (USE32) memory locations, only **EAX**, **EBX**, **ECX**, **EDX**, **ESP**, **EBP**, **EDI**, and **ESI** are valid base registers.

-----

## ALP3510: Multiple index registers

An indirect memory expression contained a combination of registers that was invalid for the selected addressing mode. More than one register was evaluated as an **index register** by the assembler; the processor only supports a single index register within **register indirect** addressing mode.

For expressions that refer to 16-bit (USE16) memory locations, only **DI** and **SI** are valid index registers.

For expressions that refer to 32-bit (USE32) memory locations, only **EAX**, **EBX**, **ECX**, **EDX**, **EBP**, **EDI**, and **ESI** are valid index registers.

-----

## ALP3511: Multiple scaling factors

In an indirect memory expression, the scaling operator (\*) was applied to more than one register term. The processor does not support more than one scaling factor within a single effective address.

-----

## ALP3512: Near target cannot be in different code segment

A JMP or CALL instruction specified a near target that was defined in a different segment, but the segments containing the instruction and the target were not named together in a GROUP directive. An instruction and its near target cannot be in different segments (addressed by the CS register) at run time.

**Recovery:** If the instruction was otherwise properly coded, use the GROUP directive to collect the segments together so that they will be accessible at run time with the same CS register value.

-----

## ALP3513: Need size for operand

The instruction operand list did not specify a size for the operation, and the assembler is unable to select a default instruction encoding because multiple variations exist.

**Recovery:** As size may be assigned to one or more of the operands by using the <type> PTR override operator.

-----

## ALP3514: Cannot establish near target without ASSUME CS:

When running the under MASM 5.10 emulation, the assembler requires the code segment register (CS) to have an ASSUME setting for the currently opened segment before it will allow the creation of explicit near code labels.

**Recovery:** Insert an "ASSUME CS:SegName" statement at the beginning of the segment before the appearance of any near code labels.

---

## ALP3515: Code generation outside of segment boundaries

A processor instruction was encountered, but no program segment has been opened.

**Recovery:** Place all processor instructions within a named program segment.

---

## ALP3516: Target is out of range by bytes(s)

The instruction references a code label or address using a self-relative displacement (a signed value relative to the address of the next instruction), but the target address requires a displacement value that is too large to be encoded into the instruction.

**Recovery:** If the SHORT operator was used in the operand field, remove it. If the instruction is of the "conditional jump" variety, it may be necessary to transform it into a "jump around a jump" by inverting the condition under which the jump is performed, then changing the target so that it references an address immediately following a "direct jump" instruction, which must be inserted and coded so that it references the original target location.

---

## ALP3517: Selected processor does not support this operand

An attempt was made to use an operand (such as a machine register) that does not exist on the processor for which code is being generated.

**Recovery:** Either use a processor selection directive to select the correct target processor, or modify the referenced operand to one that is supported on the target machine.

---

## ALP3518: Can't access data, no ASSUME or segment override

One of the following conditions occurred:

- An attempt was made to reference a named memory location that exists in a segment for which no ASSUME statement is in effect
  - An attempt was made to combine two terms in a binary expression that are relative to different segments.
- 

## ALP3519: Too many operands

This message appears during processing of an instruction operand list. More operand expressions were encountered than is valid for the instruction set of the target architecture.

**Recovery:** Remove the offending token(s) beginning at the referenced location.

---

## ALP3520: Segment address size not supported on this processor

A 16-bit processor selection directive was encountered within a 32-bit segment, or an attempt was made to reopen a 32-bit segment after switching to a 16-bit processor type. The selected processor cannot support 32-bit segments.

---

## ALP3521: Register expected

While processing an assembler directive an unexpected token was encountered at the referenced location. A processor register was expected instead.

---

## ALP3522: Selected processor does not support this instruction

A mnemonic or mnemonic/operand combination has been used that is not supported by the processor for which the assembler is currently generating object code.

**Recovery:** Verify that the correct target processor has been selected with one of the processor selection directives, or that the correct instruction form has been coded. Since ALP does not perform some of the same implicit conversions that MASM 5.10 does, use of the <type> PTR conversion operator may be required to avoid certain type-mismatch problems.

---

## ALP3523: Cannot change expression word size

After the expression word size was explicitly set with an OPTION EXPRxx directive, an attempt was made to alter the setting with another OPTION directive, or by switching to a 32-bit processor after an explicit OPTION EXPR16 was issued. Once set, the expression word size cannot be altered.

---

## ALP3601: Filename expected

The INCLUDELIB directive was used, but an error occurred while attempting to parse the filename parameter.

**Recovery:** Verify that only legal filename characters are used. If the filename appears as a quoted string literal, verify that the literal uses legal syntax according to the rules for quoted strings.

---

## ALP3602: Floating-point initializer illegal with integer variable

A floating point initializer was used on a variable that was not of type DD, DQ, DT, REAL4, REAL8, or REAL10.

---

## ALP3603: Integer initializer illegal with floating-point variable

An integer initializer was used on a variable that was of type REAL4, REAL8, or REAL10. Only floating-point initializers can be used with variables having these types.

---

## ALP3604: Expression has no effect

During a data allocation directive, an attempt was made to initialize an item using an expression that was not correctly evaluated.

**Recovery:** This may indicate an error in the internal assembler logic. Note the conditions of the error, and contact IBM.

---

## ALP3605: String is empty

During a data allocation directive, an attempt was made to initialize an item using a quoted string expression that contained no data.

**Recovery:** Quoted strings must contain at least one character value, otherwise the expression is illegal.

---

## ALP3606: Symbol "" was never defined

An identifier was declared with a GROUP or PUBLIC directive, but was never given a full definition. This condition prevents the assembler from writing the appropriate records to the output object file.

**Recovery:** Segments declared in a GROUP directive must be defined in a SEGMENT directive. Identifiers declared with the PUBLIC directive must be defined as a code label, data label, or absolute symbolic constant.

---

## ALP3607: Value not addressable

The expression following the END directive did not evaluate to segment relative address. The expression must refer to a memory location to which the operating system loader can pass control when the program is executed.

**Recovery:** Ensure that the expression contains no machine registers, and that it references a value relative to a segment defined within the module.

---

## ALP3608: Segment size exceeds 64K limit

The amount of data emitted into the current (16-bit) segment has exceeded 65536. No object file can be produced under this condition, and the current location counter has been wrapped back to zero.

**Recovery:** Reduce the amount of code or data contained in this segment, or move some of the information to another segment.

---

## ALP3701: Argument expected

While processing a directive that accepts a list of comma separated arguments, at least one argument followed by a comma was parsed, but no additional argument was encountered before the end of the line.

---

## ALP3702: Can't override array with single item

Within a structure variable instantiation, an incorrect attempt was made to override the default initializer of a structure member. The structure member was defined to be of type *array* (having been initialized with a character string or list of expressions enclosed in brackets), and the overriding expression in the structure instantiation was a single numeric expression.

**Recovery:** Array members can only be overridden using a quoted character string or a bracketed list of numeric expressions.

---

## ALP3703: .RADIX value must be one of: 2, 8, 10, or 16

Self-explanatory.

---

## ALP3704: Can't nest initializers

This message appears when a structure instantiation contained a nested override initializer within brackets, and OPTION OLDSTRUCTS was in effect. Nested structures are not allowed when the assembler is operating in this mode.

---

## ALP3705: Colon expected

Self-explanatory.

---

## ALP3706: Expecting "

When a structure or record variable is allocated, the assembler expects one or more initializer expressions to follow the structure or record type name. Initializer expressions must be enclosed in angle brackets or braces, but an unexpected token was encountered at the referenced location.

---

## ALP3707: Initializer too long or incorrect type

Within the initializer list of a structure instantiation, one of the following conditions occurred:

- An attempt was made to initialize a structure member with a character string override that exceeded the length of the member definition.
- The initializer expression type did not match that of the structure member item being initialized.

---

## ALP3708: Invalid ALIGN setting

A zero or incorrect value was specified as the argument to the ALIGN directive.

---

## ALP3709: Previous definition prevents external attribute

An attempt was made to declare an identifier as being external to this module, but a previous conflicting definition already exists. The operation is disallowed.

---

## ALP3710: Syntax error; unexpected token

The referenced token is not valid for the construct being parsed. The assembler could not attempt further processing or diagnosis of the construct.

---

## ALP3711: Previous definition prevents change in global visibility

One of the following conditions has occurred:

- The PUBLIC directive or keyword was used to export an identifier, but the identifier has already been declared with attributes that prevent it from being exported.
- A PUBLIC directive was used on a procedure name, but the PRIVATE keyword was used in the PROC directive that defined the procedure.

**Recovery:** Verify that the identifier does not appear in a COMM or EXTERN declaration, and that the identifier is a valid code or data label. Insure that the PUBLIC declaration appears before the identifier it references, and that the declaration is processed by the assembler on the first pass.

---

## ALP3712: "" must be a segment name

This message appears during processing of the GROUP directive when one of the arguments was not a valid segment name.

**Recovery:** Only identifiers defined using the SEGMENT directive are valid arguments to the GROUP directive. If the message is referencing an identifier, verify that it is indeed the name of a valid segment. Verify that the identifier was specified using the correct uppercase and lowercase letters if case sensitive assembly is in effect.

---

## ALP3713: Label outside of segment boundaries

This message appears when a label definition appears outside of any enclosing segment.

The label is an assembler alias for a segment relative machine address; such an address cannot be assigned to the label unless it appears inside of a program segment. This condition is an error, and must be corrected.

**Recovery:** Place the definition within a valid segment.

---

## ALP3714: Directive must be named

This message appears when a directive was encountered but was not preceded by a label. A label is required for this directive.

**Recovery:** Precede the directive with a valid identifier.

---

## ALP3715: Must specify all columns in .LIST ORDER

The .LIST ORDER directive did not specify a position for every possible column. All column names must appear in the list.

---

## ALP3716: Listing control stack is empty

This message appears when the user issues a .LIST POP directive and there are no listing environment entries on the stack.

**Recovery:** A matching .LIST PUSH directive must be issued before .LIST POP may be used.

---

## ALP3717: Processor mnemonic used as a label

This message is issued when a processor instruction mnemonic is used as an identifier. The severity of this message may be relaxed from Error to Warning in certain circumstances by using the **+Sk** command line switch.

Related Information:

- [Sk - Control Use of Reserved Words as Labels](#)
- 

## ALP3718: Misplaced ENDP; no open PROC

This error occurs when the ENDP (end procedure) directive was encountered, but there is no procedure currently open.

**Recovery:** The PROC directive must be used to open a procedure before the ENDP directive can be used.

---

## ALP3719: No closing bracket

An opening bracket "[" was encountered within a directive or expression, but a matching close bracket "]" was never supplied, or was misplaced.

---

## ALP3720: Data allocation outside of segment boundaries

A data definition directive was encountered, but no program segment has been opened.

**Recovery:** Place all data allocation directives within a named program segment.

---

## ALP3721: Operation illegal within structure or union

An attempt was made to use a directive or construct that is illegal within the context of a structure definition.

**Recovery:** Processor instructions are not allowed in structure definitions, and only a subset of assembler directives are legal in this context.

---

## ALP3722: Expression is not a segment or group

One of the following illegal conditions occurred:

- A segment override expression (using the colon (:) operator) did not contain a valid segment register, segment name or group name expression on the left side of the colon operator.
- An ASSUME directive contained an expression that did not evaluate to a valid segment or group name. The argument to the ASSUME directive specified a machine *segment register*, which may only be associated with a segment or group name.

**Recovery:** Verify the correct spelling of either the register argument or the segment name or group name expression.

-----

## ALP3723: ON or OFF expected

A listing control directive was encountered where the value of a flag is being manipulated; the ON or OFF keywords are the only values acceptable in this context.

-----

## ALP3724: ON, OFF, or BLANK expected

A listing control directive was encountered where the display or non-display of an individual column is being determined; the ON, OFF, or BLANK keywords are the only values acceptable in this context.

-----

## ALP3725: Phase error between passes

The address assigned to a label on pass one of the assembler had a different value on the second pass.

This usually indicates that a forward reference to a label was not fully qualified, and the eventual definition of the label was different than what was assumed by the assembler on the first pass. On the second pass, the assembler did not need to make any assumptions about the attributes of the symbol, but the resulting generation of object code caused a discrepancy in the value of the location counter.

**Recovery:** Use the listing control command line options to request a listing for both pass one and pass two of the assembler; use this listing to compare location counter values prior to the point where the phase error occurred. This will reveal the instruction that caused the location counter to become unsynchronized.

Related Information:

- [Lp - Generate Listing on Specific Pass](#)
- 

## ALP3726: Symbol already defined as different type

An identifier has been redeclared to have attributes that conflict with a previous declaration or definition.

**Recovery:** If this is an external declaration (using an EXTRN or COMM directive) referencing a data variable, ensure that the type specifier has been correctly respecified. Verify that the variable has not already been defined within this module.

External declarations for data labels or near code labels appearing within segment boundaries must not reappear within the boundaries of a different segment. Labels appearing outside of segment boundaries inherit the default address size (USE16 or USE32), and must not reappear within a segment having a conflicting address size.

Far code labels may not be redeclared with conflicting address sizes.

---

## ALP3727: PROC name mismatch

The ENDP directive was used to close the current procedure, but the name used in the ENDP directive did not match the name specified in the matching PROC directive.

---

## ALP3728: Symbol redeclared relative to different segment

A data label or near code label appearing in an external declaration was redeclared in a different segment (or outside of segment boundaries) and conflicts with a previous declaration or definition.

**Recovery:** Data labels or near code labels appearing within segment boundaries must not reappear within the boundaries of a different segment. Labels appearing outside of segment boundaries inherit the default address size (USE16 or USE32), and must not reappear within a segment having a conflicting address size.

---

## ALP3729: Attribute mismatch during reopen of segment

An existing segment was reopened using different or conflicting attributes.

**Recovery:** All identically named SEGMENT directives must be declared with the same attribute list.

If an address size attribute (USE16 or USE32) was not explicitly specified in the SEGMENT directive, verify that the default segment word size was not altered between segment declarations with a processor selection directive.

---

## ALP3730: No segment, structure, or union opened as ""

The ENDS directive was used, but no segment, structure, or union was open (or did not match the referenced name) and thus could not be terminated.

**Recovery:** If a name was given in the message, verify that it matches the name used in the associated SEGMENT, STRUC, or UNION directive. Verify that nested occurrences of SEGMENT, STRUC, or UNION are paired with a matching ENDS directive.

---

## ALP3731: Identifier expected

A directive or expression operator was used such that an identifier was expected, but none was supplied.

**Recovery:** Check for a possible misspelled identifier; Verify that the identifier was specified using the correct uppercase and lowercase letters if case sensitive assembly is in effect. Verify that the identifier is not a reserved keyword.

---

## ALP3732: Reserved symbol "" cannot be created or modified

An invalid operation was performed on a reserved identifier. One of the following conditions occurred:

- An attempt was made through an EQU (or =) directive to alter the value of a predefined identifier. Unless documented otherwise, this is an illegal operation.

- The assembler attempted the deferred creation of a reserved symbol, but a user-defined identifier already exists with the same name and has conflicting attributes.

-----

## ALP3733: Symbol redefinition error

An attempt was made to redefine an identifier in a context where redefinitions are not allowed. Redefinitions are allowed only for text macros and assembler variables assigned using the equal (=) directive.

-----

## ALP3734: Too many initializers

A bracketed expression list within a structure instantiation or record constant contained too many comma-separated expressions. The number of initializer expressions exceeded the number of elements in the structure or record definition.

-----

## ALP3735: Qualified type or type keyword expected

This message appears when a type expression or a COMM, EXTRN, or LABEL directive was expecting a type keyword, but none was supplied.

-----

## ALP3736: Unexpected text in statement

An assembler directive was fully parsed and recognized, but invalid information was encountered at the referenced location.

-----

## ALP3737:

This assembler issues this message to display user defined text when the ECHO or %OUT directives are encountered.

-----

## ALP3738: Symbol redefinition has different value

An attempt was made to redefine an EQU symbol to value which differs from a previous definition. EQU symbols may have multiple definitions only if they have identical constant values.

-----

## ALP3739: Directive illegal outside of segment boundaries

The referenced directive performs a segment-relative operation and may only be used inside of segment boundaries.

-----

## ALP3740: Alignment value must be a power of 2

The expression argument given in the ALIGN directive did not evaluate to a power of 2 (2, 4, 8, etc.).

---

## ALP3741: Alignment value not valid with current segment alignment

The referenced alignment factor was less than the alignment factor specified in the SEGMENT declaration containing the ALIGN or EVEN statement. This condition is illegal because the assembler cannot guarantee that the linker will not invalidate the requested alignment when it exercises its right to position the entire segment according to the alignment factor given in the enclosing SEGMENT declaration.

**Recovery:** Respecify either the alignment factor or the SEGMENT declaration so that the segment alignment is greater than or equal to any alignment factor requested therein.

---

## ALP3742: Redefinition has different number of fields

A RECORD type redefinition did not contain the same number of fields as a previous definition.

---

## ALP3743: Redefinition has different value

A redefinition construct was encountered (such as a type definition directive) where the redefined value did not match that of a previous definition. Redefinitions are allowed for this particular construct, but they must restate the same value given in the original definition.

---

## ALP3744: Too many bits in record definition

The referenced record definition exceeded the maximum allowable value of 32 bits.

---

## ALP3745: Record tag or fieldname expected

The operand of the MASK or WIDTH operators must be an identifier defined with the RECORD directive. This identifier must be either the tag name of the record itself, or one of the field name entries defined within the body of the RECORD directive.

---

## ALP3746: Value is out of range

The value of the referenced expression is not representable, requires too many significant bits to be stored, or lies outside the range of legal values for this context.

---

## ALP3747: Mismatch of segment address sizes in group

A group was declared to contain segments of differing address sizes. A group may contain either USE16 or USE32 segments, but not a combination of both.

---

## ALP3748: Segment already a member of group ""

An attempt was made to assign a segment to more than one parent group.

---

## ALP3749: Directive requires use of .MODEL

A simplified segmentation operation was encountered but a .MODEL directive was never processed.

---

## ALP3750: PROC must immediately precede LOCAL

A LOCAL assembler directive was encountered, and one of the following occurred:

- The LOCAL directive was not enclosed within a PROC/ENDP procedure block.
  - The LOCAL directive was positioned within a procedure block, but other assembler directives or instructions were encountered between the PROC directive and the LOCAL directive.
- 

## ALP3751: Operand has incorrect size

The size given for an operand was incorrect, or did not match that of a destination or target operand where it is to be used.

---

## ALP3752: Mismatch in attribute

The value of an attribute specified in a redeclaration or redefinition did not match the value given in the original declaration or definition. The body of the message indicates the type of attribute that is mismatched, and the assembler follows this message with two 5704 informational messages showing the coordinates and values of the mismatched constructs.

---

## ALP3753: Name collision caused by promotion from inner scope

During a definition of a structure (or union) type, a field identifier defined at the outer (current) scope had the same name as a field defined in a promoted inner scope (one created through the use of an unnamed imbedded structure). When a structure type is used to create an unnamed field within another structure type, all of the field names from the inner structure are "promoted", or made visible to the outer defining scope.

**Recovery:** One of the field identifier names must be altered to avoid the name collision. Alternatively, the unnamed imbedded structure may be given an explicit field name, in which case its own fields are no longer promoted, and fully-qualified references must be used to reach them from within expressions.

---

## ALP3754: Cannot determine calling convention

A procedure was defined to accept arguments passed on the stack by a calling routine, but no language attribute was specified or assumed for the procedure. The language attribute determines the calling convention, which in turn defines the order that arguments are pushed onto the stack.

**Recovery:** Use one of the following constructs:

- Specify an explicit language keyword in the body of the PROC directive.
- Specify a .MODEL directive with a language keyword argument.
- Specify an OPTION LANGUAGE directive.

-----

## ALP3801: Argument expected

When processing one of the EQU, IRP, IRPC, FOR, or FORC macro directives, the required argument immediately following the directive was missing or incorrectly specified.

**Recovery:** Modify the referenced token so that it is a valid argument for the directive. If this directive appears within a nested macro expansion, check to see that correct arguments were passed to outer level macros, or that outer level macro definitions are correct.

-----

## ALP3802: EXITM outside of macro

The EXITM keyword was encountered outside the context of a macro body.

**Recovery:** Verify that unexpected conditional assembly results are not affecting the block structure of the program. The EXITM keyword may not be used outside the scope of a macro body.

-----

## ALP3803: Comma expected

Self-explanatory. This message is displayed for any preprocessor directive that requires a comma where one was not supplied.

-----

## ALP3804: Extra data on line

This message appears any time the preprocessor has parsed a correctly formed preprocessor directive, but additional token(s) (other than comments) were encountered before the end of line was reached.

**Recovery:** Remove the offending token(s) beginning at the referenced location.

-----

## ALP3805: Filename expected

This message appears when an INCLUDE preprocessor directive did not contain a properly formed filename. The INCLUDE directive is ignored.

-----

## ALP3806:

This is the message printed as part of a conditional error directive; if one of these directives is processed and the user has included text information to be printed, it will appear in the <text> field of the message. If no user text was specified, this parameter will be empty and the message will contain no additional text.

**Recovery:** This was a forced error.

---

## ALP3807: Identifier expected

This message appears when a conditional preprocessor directive was expecting an identifier and one was not supplied.

---

## ALP3808: Reserved macro "" cannot be redefined

The assembler defines the referenced identifier for its own purposes; it may not be redefined by the user.

---

## ALP3809: Missing ENDM

The preprocessor was reading the body of a macro definition when the end of the current input stream was reached; the macro definition was never closed with an ENDM keyword.

**Recovery:** Verify that unexpected conditional assembly results are not affecting the block structure of the program. A macro definition may not be closed in an input stream different from the one where it was started.

---

## ALP3810: without matching IFxxx

This message indicates that an ELSE or ENDIF construct was encountered prior to encountering an IF construct.

---

## ALP3811: Reserved symbol "" cannot be modified

An attempt was made through the -D command line option to alter the value of a predefined identifier. Unless documented otherwise, this is an illegal operation.

---

## ALP3812: Symbol "" already defined

An attempt was made to define a preprocessor macro name that conflicts with an existing identifier of an incompatible type.

---

## ALP3813: expected

This message is displayed when a preprocessor directive expected a text argument enclosed in angle brackets < >, but a valid argument was

not supplied.

---

## ALP3814: Invalid character in numeric constant

The assembler was parsing a numeric constant and an alphabetic character was encountered that was not a valid radix specifier or hexadecimal digit.

---

## ALP3815: Illegal digit(s) for specified radix

A literal integer constant was qualified with radix override, but was specified using digits that are not valid for the given radix qualifier. For instance, use of the literal **1234Y** is not legal because the **Y** suffix specifies a binary number and only the digits **0** and **1** are valid binary digits.

---

## ALP3816: Expecting ">"

This message appears when the preprocessor is parsing a <text-item> and the end of line or end of file was encountered. A closing angle bracket (>) was expected.

---

## ALP3817: Unterminated COMMENT

The preprocessor was scanning the body of a COMMENT directive when the end of the current input stream was encountered; the closing delimiter character originally specified in the COMMENT directive was never encountered.

**Recovery:** Block comments may not span across input files. Provide a closing delimiter as specified in the opening COMMENT directive.

---

## ALP3896: Control character illegal in this context

A COMMENT preprocessor directive was encountered, and an attempt was made to scan for the next character which signifies the beginning and end of the comment text, but an unexpected non-printable control character was encountered instead.

**Recovery:** Only characters that are representable as printable text may be used to open and close a COMMENT sequence.

---

## ALP3897: No closing quote

The preprocessor was parsing a quoted string literal, and the end of line or other terminator was encountered before the literal was ended with a closing quote character.

**Recovery:** Verify that only single quotes (") or double quotes (") are used to open and close a string literal; they must be used in pairs. Verify that the ending quote character was not immediately preceded by another identical quote character; the assembler interprets this sequence as a request to insert a quote character into the string literal.

---

## ALP3898: Unexpected end of file

The lexical analyzer portion of the assembler preprocessor was scanning within the body of a token (for example, a block comment), when the end of the input stream was encountered.

---

## ALP3899: Unexpected terminator

The lexical analyzer portion of the assembler preprocessor was performing a text substitution operation as directed by one of the "!" or "&" operators, when the end of file or internal macro buffer was encountered.

---

## Message Numbers 4000-4999: Warning Messages

Warning messages are issued when the assembler detects a questionable construct in the input stream. The condition is not severe enough to prevent generation of an object file, but the situation should be investigated and corrected since the output program may be incorrect.

---

## ALP4201: Offset operator applied to register-indirect expression

An OFFSET operator was applied to a register-indirect expression. In MASM 5.10 emulation mode this does not cause conversion to an immediate expression; instead the register-indirect addressing mode attributes are retained, and the assembler applies the OFFSET operator to the displacement field, forcing it to have the size of the address offset. Applying the OFFSET operator to a register-indirect expression is illegal if the assembler is not operating under MASM 5.10 emulation.

---

## ALP4202: Invalid type expression; cannot convert

Under MASM 5.10 emulation, a constant numeric expression may be used as the left-hand operand of the PTR operator. In the referenced expression, the left-hand operand of the PTR operator did not evaluate to a value suitable for use as the operand size of the right-hand operand. The operand size of the right-hand operator was not converted.

---

## ALP4203: Type conversion operation has no effect

A type conversion operation involving the PTR operator was performed on an expression whose type cannot be modified. For example, the right operand of the PTR operator cannot be a register value.

---

## ALP4401: Error closing listing file "";

An error occurred while attempting to close the referenced file.

**Recovery:** Verify that no other processes are accessing the file, and that the file system is functioning correctly.

---

## ALP4402: Error deleting listing file "";

An error occurred while attempting to delete the referenced file.

**Recovery:** Verify that no other processes are accessing the file, and that the file system is functioning correctly.

---

## ALP4501: Assuming NEAR distance for operand size

A CALL or JMP instruction was coded to pass control indirectly through a memory operand of indeterminate size. When operating in MASM 5.10 emulation mode, the memory operand is assumed to have the same size as address size of the segment containing the CALL or JMP instruction, and implies a target having NEAR distance.

**Recovery:** The memory operand should be given an explicit size, regardless of whether or not the default address size and NEAR distance is the desired operation. This code will cause assembly errors if not assembled under MASM 5.10 emulation mode.

---

## ALP4502: Can't ASSUME CS to a grouped segment

An attempt was made to ASSUME the CS register to a segment that was previously named in a GROUP directive. This implies that the CS register might have different values to access the same body of code at runtime. This is illegal, and the assembler altered the ASSUME operation to refer to the group containing the segment instead.

**Recovery:** If running the assembler under MASM 5.10 emulation, change the ASSUME directive to refer to the group name containing the segment; otherwise, remove the ASSUME statement.

---

## ALP4503: Operand size does not match instruction

This is a warning message that appears when an operand specifies a size that differs from the operand size of the instruction. In this case, the operand size is implied by the instruction itself, and an explicit operand size is not required. However, if an operand size is supplied, it must match the implied size or this warning will be issued.

---

## ALP4504: [Constant] is immediate in MASM mode

This message indicates that a single expression coded as a constant value in square brackets [ ] is treated as though the brackets were not specified (when the assembler is operating in MASM emulation mode, which is the default). This warning is issued because brackets are required for an indirect memory expression when registers are involved, and the connotation is that the presence of brackets is required to force a memory reference, when in fact they are ignored.

**Recovery:** Use a segment override (for example, **DS:[1234h]**) when an indirect memory reference to an absolute address is desired. As explained above, the presence of brackets shown in the example is not required, but they are preferred for readability.

Note: A future release may provide an alternate mode of operation such that bracketed constant values will be treated as memory references; this warning message thus points out constructs that are incompatible with any future releases operating in this mode. Since the presence of brackets are currently redundant in this context (and indicate a possible programming error), it is recommended that they be removed.

---

## ALP4505: Access to data through a code label

This warning occurs when an instruction attempts a memory access through a code label (a procedure name or a label followed by a colon). This is an invalid operation unless a type conversion is first performed on the expression containing the label.

---

## ALP4506: Selected processor does not support this instruction

This message is issued under the same circumstances as message ALP3522, but has reduced severity when the assembler is operating under MASM emulation.

-----

## ALP4507: Only storing NEAR portion of FAR pointer

Within a data allocation directive, a variable was initialized to contain the address of a FAR code label or variable defined in another segment. However, the size of the variable being initialized is not large enough to hold the fully qualified address (both segment and offset) of the item, and only the offset portion was stored.

**Recovery:** If the full address of the pointer is desired, then the size of the data item being initialized must be increased. Otherwise, the OFFSET operator should be used in the address expression to truncate the segment information and suppress this warning.

-----

## ALP4508: Operand size inferred from immediate value

When operating under MASM 5.10 emulation mode, the assembler allows an immediate value to determine the size of the memory operand to which it is applied if its magnitude exceeds that which will fit into a byte. In this case, it is assumed that the operation refers to a word-sized memory operand (2 bytes in USE16 segment, 4 bytes in a USE32 segment). If the magnitude of the immediate value is sufficiently small (less than 128), then the operation is ambiguous, and an error is generated because the assembler does not know whether to treat the memory operand as a byte or a word value.

**Recovery:** An explicit size should be given to the memory operand. Code relying on this behavior will not assemble correctly if the assembler is not operating in MASM 5.10 emulation mode.

-----

## ALP4509: Operand size mismatch

One of the following conditions occurred:

- An attempt was made within a data allocation directive to initialize an item with an expression having an explicit and different size.
  - A memory operand with an explicit size was used in conjunction with a register operand, but two operand sizes did not match. The register operand size overrides the size of the memory operand.
  - A memory operand with an explicit size was used in conjunction with an address offset. The size of the memory operand and the size implied by the address size of the offset value did not match.
- 

## ALP4510: Truncation of significant bits in immediate value

One of the following conditions occurred:

- An operand expression was encoded into the instruction as an immediate value, but the magnitude of the numeric expression exceeded the number of bits required to store it in instruction encoding; the value was truncated to fit in the allotted space.  
  
Since the assembler uses 32-bit arithmetic during expression processing, operations such as negation or logical inversion of small numeric quantities can result in values that require all 32 bits of precision.
- A relocatable immediate expression was converted to a smaller type, thus affecting the type of the relocation record generated for resolution by the linker. This could happen if an offset expression was forced into a byte sized storage space.

**Recovery:** Use the <type> PTR override to explicitly convert the expression to a value of the proper size.

---

## ALP4511: Assuming segment width for operand size

This message can occur when a memory operand or an immediate operand of indeterminate size is used as an operand to a PUSH or POP instruction. When operating in MASM 5.10 emulation mode, the assembler assumes the operation to involve an operand size which is equal to that of the address size of the enclosing segment.

For example, when using a 32-bit processor, the following instruction:

**PUSH DS:[0]**

is ambiguous because the processor can retrieve either a 16-bit or 32-bit value from memory location 0.

**Recovery:** Use a memory expression that has an explicit operand size. For example:

**PUSH DS:[DWORD PTR 0]**

---

## ALP4512: Can't ASSUME CS to different segment or group; ignore

Within an open segment, an attempt was made to ASSUME the CS register to a different segment, or to a group not containing the currently opened segment. The operation was not allowed.

**Recovery:** Remove the ASSUME statement, or adjust it so that it specifies the currently opened code segment or a group containing the segment.

---

## ALP4513: Invalid mnemonic/operand combination

This message is issued under circumstances similar to that of message ALP3505. In this case an instruction encoding was found for the given mnemonic/operand combination, but the encoding is considered invalid when the assembler is not operating under this level of MASM emulation; thus this warning is issued.

---

## ALP4514: No size for operand, assuming default

This message indicates that no operand size was given for the instruction, but the assembler was able to infer an operand size from the instruction itself. This message is only issued when the assembler is operating under MASM 5.10 emulation. In this mode, a default operand size is associated with certain processor mnemonics; this default value is used when no explicit operand size is given.

**Recovery:** An explicit operand size should be given for the referenced expression because the assembler is automatically resolving a potentially dangerous ambiguity in its selection of a default operand size.

---

## ALP4601: Error closing object file "";

An error occurred while attempting to close the referenced file.

**Recovery:** Verify that no other processes are accessing the file, and that the file system is functioning correctly.

---

## ALP4602: Error deleting object file "";

An error occurred while attempting to delete the referenced file.

**Recovery:** Verify that no other processes are accessing the file, and that the file system is functioning correctly.

---

## ALP4603: Missing END

The end of file was encountered but an END directive was never processed. All top-level assembler modules invoked from the command line must have an END directive as the last statement in the file. Files processed with the INCLUDE preprocessor directive should not contain an END statement.

---

## ALP4701: Unterminated PROC

When the end of the source input stream was encountered, it was determined that a procedure was opened with a PROC directive, but never closed.

**Recovery:** Any procedures opened with PROC must be closed within the same input stream using the ENDP directive.

---

## ALP4702: Unterminated segment ""

When the end of the source input stream was encountered, it was determined that a segment was opened with a SEGMENT directive, but never closed.

**Recovery:** Any segments opened with SEGMENT must be closed within the same input stream using the ENDS directive.

---

## ALP4703: Unterminated structure or union

When the end of the source input stream was encountered, it was determined that a structure or union was opened with a STRUC/STRUCT or UNION directive, but never closed.

**Recovery:** Any structure or union must be terminated using the ENDS directive.

---

## ALP4704: Address size mismatch

The referenced address expression was used as a source operand, but the address size of the expression did not match the size of the target operand. An implicit conversion was applied to the address size of the referenced expression.

---

## ALP4706: Processor mnemonic used as a label

This message is issued when a processor instruction mnemonic is used as an identifier. This condition is normally an error if the **+Sk** command line switch is turned off.

Related Information:

---

## ALP4707: Alignment value not valid with current segment alignmen

The referenced alignment factor was less than the alignment factor specified in the SEGMENT declaration containing the ALIGN statement. This condition is illegal because the assembler cannot guarantee that the linker will not invalidate the requested alignment when it exercises its right to position the entire segment according to the alignment factor given in the enclosing SEGMENT declaration.

**Note:** This condition is allowed to exist as a warning under MASM 5.10 emulation for backward compatibility with existing source files.

**Recovery:** Respecify either the alignment factor or the SEGMENT declaration so that the segment alignment is greater than or equal to any alignment factor requested therein.

---

## ALP4708: Symbol redeclared relative to different segment

This message is issued under the same circumstances as message ALP3728, but has reduced severity when the assembler is operating under MASM emulation.

---

## ALP4709: Label outside segment boundaries

This message is issued under the same circumstances as message ALP3713, but has reduced severity when the assembler is operating under MASM 5.10 emulation.

---

## ALP4710: Identifier expected

This message is issued under the same circumstances as message ALP3731, but has reduced severity when the assembler is operating under MASM 5.10 emulation.

---

## ALP4711: Attribute respecification ignored

An attempt was made in a directive to change an attribute that has already been specified. Once set, the attribute cannot be changed. If the attribute is respecified, it must match the existing setting or this warning will be issued.

---

## ALP4712: Mismatch of segment address sizes in group

This message is issued under the same circumstances as message ALP3747, but has reduced severity when the assembler is operating under MASM 5.10 emulation.

---

## ALP4713: Attribute mismatch during reopen of segment

This message is issued under the same circumstances as message ALP3729, but has reduced severity when the assembler is operating under MASM 5.10 emulation.

-----

## ALP4801: Identifier expected, condition is false

For compatibility with MASM, the IFDEF and IFNDEF preprocessor directives will accept a non-identifier token as an argument, but this warning is issued to indicate that the condition cannot be properly tested, and the result is always false.

**Recovery:** Modify the argument so that it is a correctly formed identifier.

-----

## ALP4802: Extra data on line

This message appears under the same conditions as that of ALP3804, but for better compatibility with MASM the severity has been reduced from an error to a warning. This message will be issued if an IFDEF, IFNDEF, ELSEIFDEF, or ELSEIFNDEF directive contains extra data at the end of the line.

**Recovery:** Remove the offending token(s) beginning at the referenced location.

-----

## ALP4803: Expecting ">"

This message appears when the preprocessor is parsing a <text-item> and the end of line or end of file was encountered. A closing angle bracket (>) was expected.

-----

## ALP4804: Expression expected, zero assumed

For compatibility with MASM, certain preprocessor directives that expect an expression operand will not issue an error if the operand is absent. This warning is issued instead to indicate that an implicit expression value of zero is assumed.

-----

## ALP4899: Illegal character

This message is issued by the text preprocessor when a character was encountered in the source stream that is not part of the valid execution character set.

**Recovery:** This may cause undefined behaviors, and a text editor should be used to remove the offending character.

-----

## Message Numbers 5000-5999: Informational Messages

Informational messages may be requested with a command line option (see [M - Control Individual Messages or Groups](#)) and provide the user with a variety of useful information. All informational messages are disabled by default.

-----

## ALP5001: Number of Errors :

Informs the user of the number of error messages issued during the assembly.

-----

## ALP5002: Number of Warnings :

Informs the user of the number of warning messages issued during the assembly.

-----

## ALP5003: Number of Symbols :

This message indicates how many user identifiers were defined during the assembly.

-----

## ALP5101: Opened message output file ""

This message indicates that the message output processor has opened and is prepared to write to the referenced file.

-----

## ALP5201: Operand is declared relative to ""

This message indicates that the referenced relocatable expression is declared relative to the given segment or group name.

-----

## ALP5301: Assembler is on source pass

This messages indicates that the assembler has begun processing the referenced pass through the input stream.

-----

## ALP5401: Closed listing output file ""

This message indicates that the listing file processor has closed the referenced file.

-----

## ALP5402: Deleted listing output file ""

This message indicates that the listing file processor has deleted the referenced file.

-----

## ALP5403: Opened listing output file ""

This message indicates that the listing file processor has opened and is prepared to write to the referenced file.

---

## ALP5501: Address size is

This message provides information to supplement the occurrence of other related warning or error messages. The expression referenced by the message coordinates has inherited or has been assigned the address size given in the message text.

---

## ALP5502: No size for operand, assuming default

This message indicates that no operand size was given for the instruction, but the assembler was able to infer an operand size from the instruction itself.

---

## ALP5503: Instruction padded with NOP(s)

This message indicates that the assembler generated one or more NOP instructions to follow the object code generated for the referenced instruction. The instruction operand list contains a forward-referenced expression which forced the assembler to allow space for the longest possible instruction encoding on the first pass. The generation of NOP instructions may be avoided by qualifying the forward reference with a <type> PTR override.

---

## ALP5504: Operand size is

This is an informational message that typically accompanies other errors or warnings dealing with operand size problems. This message is issued for every operand expression involved in the condition that caused the associated error or warning message, and indicates the operand size of the referenced expression.

---

## ALP5505: Segment override has no effect

This message is issued when a segment override operator was used in an expression, but the override was discarded because the offset operator was applied.

---

## ALP5506: Generated ASSUME CS:

This message is issued when an attempt is made to generate code in an open segment for which there is no valid ASSUME CS in effect. The assembler automatically generates an ASSUME statement such that the CS register is associated with the currently opened segment or with the group containing that segment if one exists.

---

## ALP5601: Closed object output file ""

This message indicates that the object file processor has closed the referenced file.

---

## ALP5602: Deleted object output file ""

This message indicates that the object file processor has deleted the referenced file.

-----

## ALP5603: Opened object output file ""

This message indicates that the object file processor has opened and is prepared to write to the referenced file.

-----

## ALP5701: Assembly terminated by .ABORT

The .ABORT directive was used to terminate the assembler at the referenced location.

-----

## ALP5702: Address size assumed to be

This messages indicates that the referenced identifier was an external code label declared outside of segment boundaries, and was assumed to be defined in an external segment having the referenced address size.

-----

## ALP5703: Structure redefinition: ""

This message indicates that the current structure definition is replacing a previous definition of the same name.

-----

## ALP5704: Declaration sets =

This message describes the attribute that is mismatched in a segment redeclaration.

-----

## ALP5705: outside of segment boundaries

This message appears when an external declaration (COMM or EXTRN) appears outside of any enclosing segment.

If an external declaration is not enclosed in a segment definition that describes how the external symbol is ultimately defined, the assembler is deprived of segment attribute information; in particular, USE16 versus USE32. This could cause the assembler to generate incorrect object code, and may also cause linker errors.

**Recovery:** Place the external declaration within a segment definition that correctly reflects the segment definition in the external module where the symbol is defined.

-----

## ALP5801: Begin skipping tokens

This message indicates that the results of a conditional assembly directive were evaluated to be false, and that the preprocessor has begun discarding tokens at the referenced location. No tokens will be returned to the parser until an appropriate ELSE or ENDIF condition is encountered.

---

## ALP5802: Finished skipping tokens

This message indicates that a false conditional block was ended with an appropriate ELSE or ENDIF construct. The preprocessor will begin returning tokens to the parser after the referenced location.

---

## ALP5803: Closed source file ""

This message indicates that the preprocessor has closed the referenced file.

---

## ALP5804: Opened source file ""

This message indicates that the preprocessor has opened and is prepared to read from the referenced file.

---

## ALP5805: Macro redefinition: ""

This message indicates that the current macro definition is replacing a previous definition of the same name.

---

## ALP5806: Opened INCLUDE file ""

This message indicates that the preprocessor has opened and is prepared to read from the referenced include file.

---

## ALP5807: Closed INCLUDE file ""

This message indicates that the preprocessor has closed the referenced include file.

---

## ALP5901: Closed response file ""

This message indicates that the command line processor has closed the referenced file.

---

## ALP5902: Opened response file ""

This message indicates that the command line processor has opened and is prepared to read from the referenced file.

---

## Return Codes

When ALP completes, it passes a return code back to the program that invoked it. This return code shows whether ALP completed successfully or with an error.

The return codes are:

0	Normal program completion.
1	User-specified file not found.
2	Unexpected system error.
3	Terminated by user or operating system.
4	Syntax errors in input file.
5	Command line usage error.
6	Internal sanity check failure.
7	Error accessing ALP messages file.

---

## Notices

*February 2000*

**The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS". WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Requests for technical information about IBM products should be made to your IBM authorized reseller or IBM marketing representative.

(C) Copyright International Business Machines Corporation 1995-2000. All rights reserved. Note to U.S. Government Users -- Documentation related to restricted rights -- Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

The [Processor Reference](#) portion of this manual contains information reprinted with permission from Intel Corporation.

---

## Disclaimers

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights or other legally protectable rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood NY 10594, U.S.A.

---

## Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries:

IBM  
Operating System/2  
OS/2  
Presentation Manager

The following terms are trademarks of other companies:

Microsoft - Microsoft Corporation  
Pentium - Intel Corporation  
Pentium Pro - Intel Corporation  
UNIX - UNIX System Laboratories, Inc.

-----